

Domain-specific Visual Languages for Model-driven Software Engineering

A THESIS PRESENTED

BY

JOHN C. GRUNDY

PHD (1994, UNIVERSITY OF AUCKLAND),

MSc (1991, UNIVERSITY OF AUCKLAND),

BSc(HONS) (1989, UNIVERSITY OF AUCKLAND)

IN FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF SCIENCE

UNIVERSITY OF AUCKLAND, AUCKLAND, NEW ZEALAND

AUGUST 2021

Abstract

THIS THESIS IS A COLLECTION OF MY ORIGINAL SCHOLARLY WORKS published in peer reviewed journals and conferences. The collection of articles investigate the intersection of the fields of Domain-Specific Visual Languages (DSVLs) and Model-driven Engineering (MDE). These combined allow software engineers – and in limited circumstances end users of software systems – to specify complex software system models at high levels of abstraction (using DSVLs), and then use these models to generate parts (or even all) of the modelled target software system code, configurations, user interfaces, data formats, test cases, and/or other implementation-level details (using MDE).

The first set of articles discuss the development of a range of novel DSVL and MDE supporting tools. The second set of articles show how these can be used to support software engineers to conduct requirements engineering and define software architectures for complex software systems. The third set of articles discuss support for software engineers in designing, implementing and testing software systems using DSVLs and MDE. The fourth set of articles present DSVL and MDE-based approaches to supporting software process management. The fifth set of articles present a variety of “human-centric” and collaborative approaches to supporting these tasks in DSVL-based tools. The sixth set of articles describe support for DSVL and MDE-based tools targeted to “end users”, allowing these non-technical end users to define and generate their own software solutions. I conclude with a recent article describing future directions for the field.

FOR JUDITH, STEPHANIE, JESSICA, JOSHUA, ALEXANDER AND HANNAH.

Acknowledgments

I WOULD LIKE TO GREATLY THANK my wife Judith for her many years of love for me and support for my professional work, but also for my family life which gives my work meaning. My children Stephanie, Jessica, Joshua, Alexander and Hannah I would like to thank for their love and support, and also their understanding when work demands, especially overseas and interstate travel, have taken me away from them and sometimes important things happening in their lives.

Special appreciation goes to my PhD supervisor and mentor Professor John Hosking, who I have continued to work and co-author with to this day, as evidenced by a number of co-authored works in this thesis. There are a great many others I would like to thank for their support of my work and our valuable collaborations. These include, but are certainly not limited to, Rick Mugridge, Robert Amor, Mohamed Abdelrazek, Iman Avazpour, Amani Ibrahim, Jean-Guy Schneider, Emilia Mendes, Hourieh Khalajzadeh, Tanjila Kanij, Rashina Hoda, Qiang He, Yun Yang, Feifei Chen, Norsaremah Salleh, Massila Kamalrudin, Svetha Venkatesh, Xin Xia, David Lo, Li Li, Scott Barnett, Rajesh Vasa, Ewan Tempero, and many others. Most of the research I have advanced has been done in collaboration with the great many students I have been fortunate enough to supervise, including most of the works collected here.

The significance of this body of work has been recognised by my award of a very prestigious Australian Research Council Australian Laureate Fellowship in 2019 for the project “Human-centric, Model-driven Software Engineering” – a direct result of this long programme of work on DSVLs and MDE. Prior recognition of my research leadership and contributions has included an Alfred Deakin Professorship (2017), and Fellow of Automated Software Engineering (2012), awarded by the Steering Committee of the Automated Software Engineering conference, where some of these works included in the thesis were published.

I would like thank the funding bodies for appreciating and supporting the work I do, including in particular the Foundation for Research, Science and Technology and the Australian Research Council, the majority of the works in this thesis supported by these two funders. I greatly appreciate the many companies I have worked with on a wide range of challenging, interesting and forward-looking R&D projects, several of which are also reported in chapters in this thesis. This includes work with Orion Health, XSOL, PRISM, CSIRO, CA Labs, Peace Software, Thales, NICTA, Sofismo, and others.

Finally I thank the many supervisors I have had for supporting my research endeavours, sometimes when they conflicted with other Department, Faculty and University needs, including Mark Apperley, John Hosking, Peter Brothers, Michael Davies, Leon Sterling, John Wilson, Peter Hodgson, Jon Whittle and Ann Nicholson.

Contents

1	INTRODUCTION	3
1.1	Visual Modelling Languages in Software Engineering	3
1.2	Domain-Specific Visual Languages in Software Engineering	4
1.3	Model-Driven Engineering of Software	11
1.4	DSVLs and MDE for Software Engineering	11
1.5	Overview of the papers in this Thesis	19
1.6	Evidence of Impact	26
	REFERENCES	33
2	DSVL MODELLING TOOL DEVELOPMENT	34
2.1	Constructing component-based software engineering environments: issues and experiences	34
2.2	Inconsistency Management for Multi-view Software Development Environments	45
2.3	Pounamu: a meta-tool for exploratory domain-specific visual language tool development	68
2.4	Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications	90
2.5	VikiBuilder: end-user specification and generation of Visual Wikis	124
3	DSVLs AND MDE FOR SOFTWARE REQUIREMENTS AND ARCHITECTURES	135
3.1	Aspect-oriented Requirements Engineering for Component-based Software Systems	135
3.2	MaramaAIC: Tool Support for Consistency Management and Validation of Requirements	144
3.3	Adaptable, Model-driven Security Engineering for SaaS Cloud-based Applications	182
3.4	SoftArch: tool support for integrated software architecture development	224
3.5	A Visual Language for Design Pattern Modelling and Instantiation	244
4	SOFTWARE DEVELOPMENT AND TESTING USING DSVLS AND MDE	266
4.1	Supporting Multi-View Development for Mobile Applications	266
4.2	Specifying Model Transformations by Direct Manipulation using Concrete Visual Notations and Interactive Recommendations	292
4.3	SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions	333
4.4	Realistic Load Testing of Web Applications	358
4.5	A Domain-Specific Visual Modeling Language for Testing Environment Emulation	370
5	SOFTWARE PROCESS MANAGEMENT WITH DSVLS AND MDE	380
5.1	Serendipity: integrated environment support for process modelling, enactment and work coordination	380
5.2	A decentralized architecture for software process modeling and enactment	410
5.3	Collaboration-Based Cloud Computing Security Management Framework	422
5.4	DCTracVis: a system retrieving and visualizing traceability links between source code and documentation	431
5.5	An End-to-End Model-based Approach to Support Big Data Analytics Development	470
6	HUMAN-CENTRIC DSVL MODELLING AND COLLABORATION	498
6.1	Experiences developing architectures for realising thin-client diagram editing tools	498

6.2	Engineering plug-in software components to support collaborative work	527
6.3	A generic approach to supporting diagram differencing and merging for collaborative design	555
6.4	A 3D Business Metaphor for Program Visualization	566
6.5	Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool	573
7	END-USER APPLICATIONS OF DSVLS AND MDE	584
7.1	Domain-specific visual languages for specifying and generating data mapping systems . . .	584
7.2	A domain-specific visual language for report writing	602
7.3	Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-driven Engineering	611
7.4	A visual language and environment for enterprise system modelling and automation	620
7.5	A suite of visual languages for model-driven development of statistical surveys and services .	650
7.6	Engineering Complex Data Integration and Harmonization Systems	681
8	FUTURE DIRECTIONS	714
8.1	Towards Human-Centric Model-Driven Software Engineering	714

1

Introduction

Software engineering depends on models of varying levels of abstraction e.g. software processes, requirements, architecture, UI and database design, code, test cases etc. Because of their complexity, software engineers have developed visual representations of these models, as diagrams, over the past several decades. Many of these diagrammatic visual models are general-purpose and fit for modelling various aspects of most software systems. Examples include flow diagrams [13], state transition diagrams [62], entity relationship diagrams [21], state charts [42], data structure diagrams [9], and various object-oriented modelling languages, such as the Unified Modelling Language [22]. While some of these techniques are limited to certain aspects of software system modelling, in general they are designed to model “any” (or most) kinds of software application for any (or most) kinds of software domains of application.

An alternative approach to these “general purpose” software visual modelling languages is to use what I term “domain specific visual languages” – or “DSVLs” – models suited to a (sometimes very) limited domain of software systems and for (sometimes very) limited modelling tasks. These DSVLs sacrifice generality but have one or more advantages over general purpose modelling languages for this specific domain. These include using higher levels of abstraction, fitting more closely to the modeller’s cognitive model of their domain and its software, and/or use of iconic or visual structures familiar to the modeller in the domain of application. Different DSVLs might be used by all software engineers or by those working on specific phases of development or in specific problem domains. They might even be designed for use by non-technical “end users” of software systems to allow them to – under constrained conditions – model their own software.

Software models must typically be translated by hand into implementations using programming language source code and/or other implementation-level artefacts, such as data schema, user interface elements, business logic and processing code, configuration files, and so on. An alternative approach is to use “Model-Driven Engineering” – or “MDE” – where a **tool** automatically translates one or more high level models into lower-level models and/or implementation-level artefacts. Advantages of such MDE approaches can include much faster software implementation, improved quality of software, enabling less technically proficient developers to realise solutions, and even allowing end users to model and build their own software solutions [66]. MDE may use general-purpose models, textual domain-specific languages, or models created and visualised with domain-specific visual languages.

This thesis summarises my efforts over many years to develop domain-specific visual language (DSVL) for software engineering models and associated model-driven engineering (MDE) tools to turn these models into software solutions.

1.1 Visual Modelling Languages in Software Engineering

Visual Languages have been used for thousands of years to communicate complex ideas, or “models” of the world. They use a range of human capabilities and allow complex models to be presented and manipulated

using metaphors close to a user's cognitive models. The idea is to provide a way to visualise and/or author complex models using human visual capabilities, compared to textual representations [60, 68, 59]. These include but are not limited to boxes and lines, information containment, colour and shading, formal or informal annotations, sketched or computer-constructed diagrams, iconic representations, and textual labels [12, 59].

Software Engineering has depended on using visual models to represent complex software system characteristics even before software engineering as a discipline in its own right began [59, 55]. Even a non-trivial software system is made up of many concepts, ranging from high level requirements describing the problem space; architectural abstractions describing both software and hardware components in the solution space; design-level data, interface, processing logic and other information about decisions made about implementing the software; various information about testing and deployment; and are often used to aid in software process implementation and project management. Such visual models can be constructed manually to aid development, might be reverse-engineered from existing code and other implemented system artefacts, or might be generated from other, higher level models [66, 6].

As software systems have grown ever more complex, new general-purpose visual modelling techniques have been developed to help software engineers to manage this complexity and model a very diverse range of systems. Entity-relationship diagrams are still a very commonly used approach to describe database structures [21]. Various forms of state transition diagrams and state charts model how software system state changes over time [62, 42]. The Business Process Modelling Language (BPML) [70] has been widely used to model general business processes, technical implementations of such processes as services, and service communication. The Unified Modelling Language (UML) [22] has been used to model software requirements, system designs, software designs, and various specialised aspects of systems. Computer Aided Design (CAD) tools have been deployed in many engineering domains and many share common representations of models [43].

Key advantages of such general purpose visual modelling approaches include:

- standardised modelling constructs, semantics and notations;
- notations familiar to a wide range of users in the general domain;
- 2D (and sometimes 3D) layouts and visual elements allowing wide range of model representations;
- tool support to build models, reverse engineer models from code, and support aspects of model-driven engineering; and
- useful for solving problems in a very wide variety of application domains.

1.2 Domain-Specific Visual Languages in Software Engineering

Such general-purpose modelling languages are not always the best choice to use to model a software system. They do not leverage particular domain-specific concepts, leading often to overly-complex models, cluttered visual formalisms, hard-to-read and hard-to-maintain diagrams [17, 67, 72, 37]. These problems are caused by using modelling and notational concepts designed for a very wide range of purposes. In order to describe a problem in a target domain, a large number of general-purpose abstractions may need to be assembled. No domain-specific concepts are built into the modelling language and hence models need to be composed to describe them. General purpose visual notations similarly need to be composed or augmented in often cumbersome ways to express domain-specific concepts e.g. with UML profiles and BPMN annotations. Usability can become a major problem of both the general purpose visual notation and its supporting tools [59, 1, 64]. Even domain-specific extensions to the general purpose modelling language, such as UML profiles and BPMN annotations, do not solve these issues, and often further complicate the visual models used [65, 64, 14].

Domain-specific Visual Languages (DSVLs) provide a powerful, human-centric approach to presenting and manipulating complex information [58, 41, 20, 51, 69]. DSVLs are designed for use in a specific domain with a domain-specific set of modelling concepts and a domain-specific set of visual notational elements [58, 51]. As domain concepts are built into the language, modelling problems in the domain – for which

the DSVL is targeted – typically requires much smaller models and visual notational elements. Ideally the DSVL visual notation leverages both concepts and representations – “visual metaphors” – from its target domain of use. This makes the DSVL both more efficient for modelling in the target domain, but also better suits its target end users’ modelling needs than a general purpose modelling language.

1.2.1 What are DSVLs?

There is no specific, generally accepted definition for a “DSVL”. In fact, they go by a variety of names e.g. visual domain-specific language, domain-specific visual modelling language, or domain-specific visual model. I define a Domain-specific Visual Language (DSVL) – for the purpose of this thesis and based on a definition John Hosking and I devised many years ago – to be:

“a visual modelling language where the model and notation are customised for a particular problem domain”

In DSVLs we make a trade off between generality of the language – i.e. the range of problems that are able to be solved – and terseness of notation and closeness of mapping to the target problem domain – i.e. how specialised the visual notation is for use in a particular domain. By this I mean that we have a specialised visual modelling language only suitable for use in specific application domains, **but which is optimised for these domains.**

The critical features of a DSVL are thus:

- an underlying model – or “meta-model” – with constructs for modelling only within the target domain, not very general problem domains;
- visual notations ideally familiar to the DSVL target users in the specific target domain;
- the use of visual metaphors specific to the target domain;
- only useful (usually) for solving problems in the target domain; and
- sometimes is an existing, preferred modelling language for its users in the target domain.

To give tangible examples of such DSVLs, I briefly review a few below, with their key features, advantages – and some key limitations. Enterprise Modelling Language [52], Horus-HPC [3] and Statistical Design Language [50] are described in detailed chapters of their own later in this thesis.

ENTERPRISE MODELLING LANGUAGE (EML)

EML (Enterprise Modelling Language) is a DSVL that we invented for modelling business processes and process structures [52]. It uses a novel tree-based metaphor to structure services within an organisation into a hierarchical form, similar to that used to group organisational structures. A novel overlay metaphor is used to describe process flow, linking services in the hierarchy together to form the process flow. Multiple overlays correspond to different processes. Process branching can be visualised by diverging or converging overlaps. Exceptions and error handling can be described with specialised overlays. A supporting tool, EMLTool [52], provides an authoring tool with novel fish-eye viewer and other large-scale DSVL support.

Figure 1.1 shows a student enrolment system’s key services and processes modelled in EML. This example shows a tree structure being used to represent different components and work tasks implemented as discrete services in a student management system, overlaid with multiple “process flows”. The tool, EMLTool, allows users to show and hide different process overlays, break large enterprise service hierarchies into multiple views, and provides large scale service representation via fish-eye views and alternative views of processes using a subset of BPML. Business Process Execution Language for Web Services (BPEL4WS) scripts are generated from the EML models to be “run” by a BPEL-based workflow management system.

EML does not support general software application modelling and only a subset of service orchestration. It is oriented towards knowledgeable domain experts who are non-technical users of complex service-oriented applications and want and need the power to configure new process flows using pre-implemented and hosted services. While EML supports flexible and scalable modelling of these services, services themselves can not

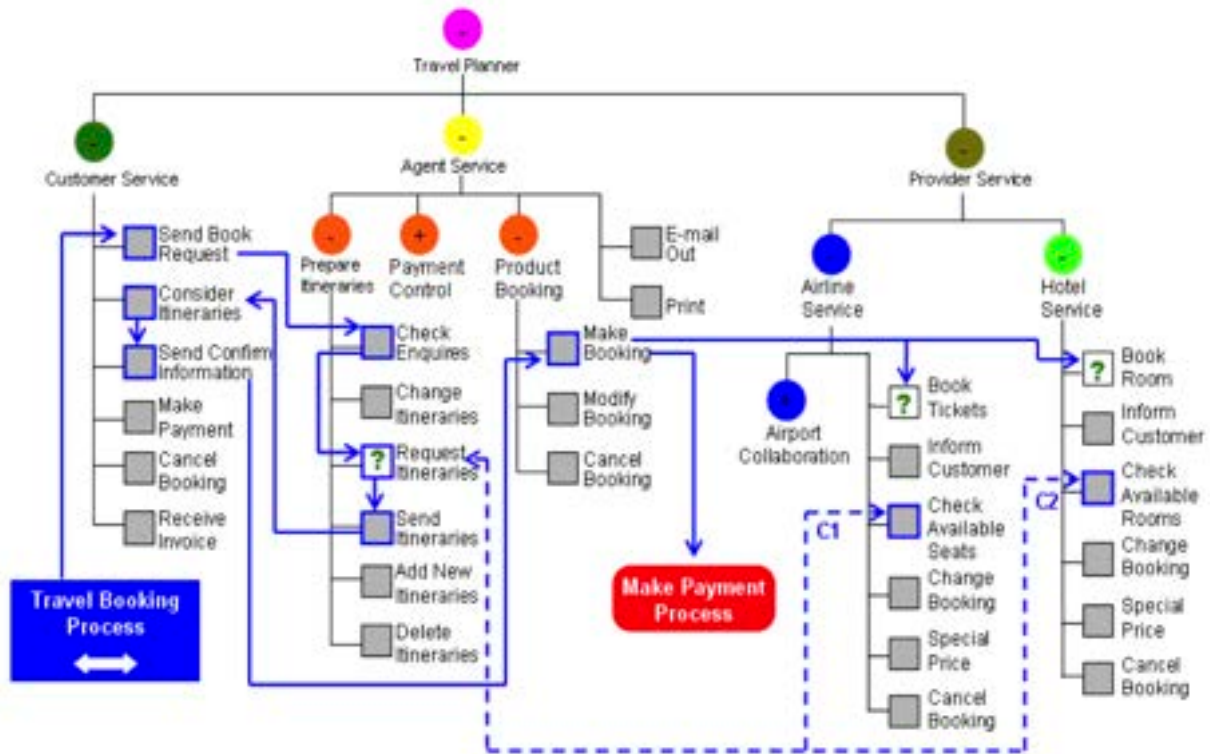


Figure 1.1: Enterprise Modelling Language (EML) showing a student enrolment process (from [52]).

be built, only aggregations and orchestrations. No data storage and user interface capabilities for the services are provided by EML itself. The tree-and-overlay metaphor is natural for these target end users but may be frustrating or counter-intuitive to others, including service implementers.

HORUS-HPC

Horus-HPC (High Performance Computing) is a web-based IDE for developing parallel programs for complex scientific tasks to be run on GPUs, CPU grids or cloud computing platforms [3]. The target end user is a scientific programmer who is knowledgeable about the target scientific problem domain; has some coding skills; and some limited HPC design and development skills. The idea is to use a set of DSLs at differing levels of abstraction to model target problem domain (e.g. molecular simulation or pulsar discovery algorithms), including scientific formulae; model sequential solutions; model - with the help of packaged patterns - HPC parallelised solutions; describe deployment onto a target HPC platform; and generate parts of a C or GPU kernel program code implementation; and then complete this code by hand. The tool is not designed for complete code or script generation but to be a quality and productivity improvement support platform. The DSLs have the added advantage that the higher level ones are aimed at mirroring how scientific end users actually describe problems in their domain using whiteboards, formulae, pseudo-code, schematics etc.

Figure 1.2 shows an example of three Horus-HPC diagrams modelling a parallel program. The left hand side diagram shows a high level box-and-line representation of key HPC program components wired together. The top right formulae editor allows scientists to model mathematical aspects of their problems at a high level and link these abstractions both to box-and-line component realisations and code implementations of (parts of) their scientific problem. The C code editor in the bottom right shows a generated program from the component diagram with user additions of code. The message passing (OpenMPI) or GPU (OpenCL) code is generated from lower-level component diagrams that map the algorithm components onto HPC hardware implementation components for the chosen implementation platform. Further views include data visualisation and modelling to show results of complex calculations as charts, scatter plots, etc.

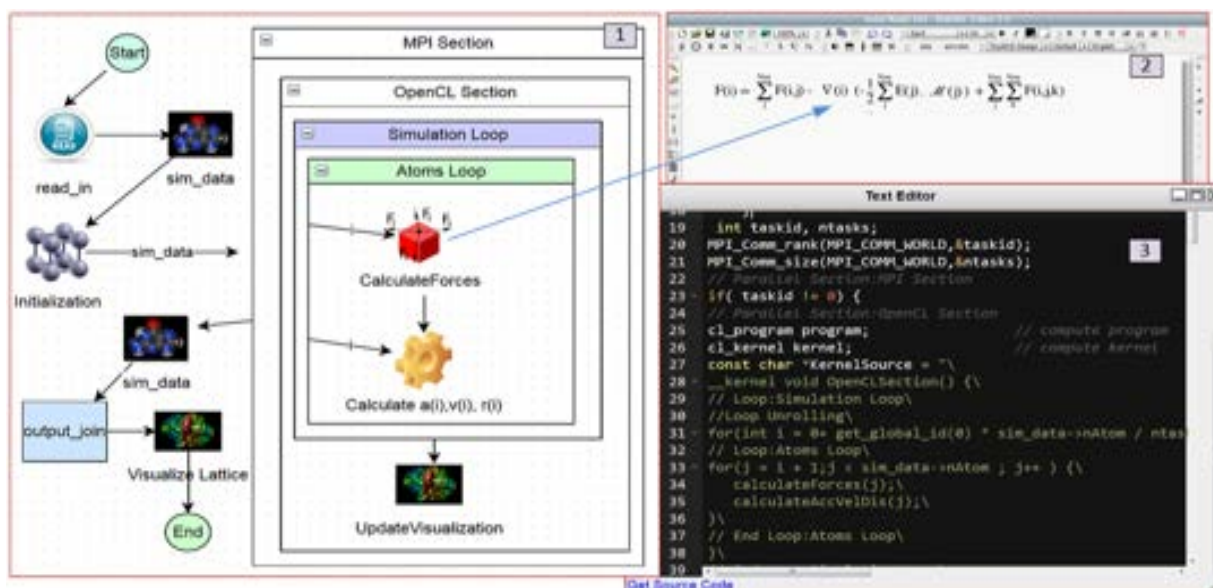


Figure 1.2: Horus-HPC being used to model a parallel program (from [3]).

and to model complex input and output data formats used to produce the generated C code implementation data structures.

While Horus-HPC provides a fully-fledged, cloud-based development environment for a very wide range of HPC applications and HPC hardware platforms, removing the 3GL C code editor from the IDE leaves it with special purpose modelling views that are not general purpose. These have been designed for scientist-programmers to model several levels of abstractions of their HPC programs. While their models allow us to generate significant amounts of HPC C code, they do not generate full implementations without direct textual C code editing.

STATISTICAL SURVEY DESIGN LANGUAGE (SDL)

SDL (Statistical Design Language) is a DSVL that we invented for designing and enacting statistical surveys. It provides a suite of related visual modelling languages addressing different aspects of statistical survey design and implementation using the R statistical analysis tool [50]. Survey diagrams are used to represent a high level view of a survey including purpose, key population sources and key outcomes. Survey data diagrams are used to model the datasets used in the surveying including data sourcing and management. Task diagrams are used to model the key steps involved in running the survey including pre-survey activities and post-survey activities. Technique diagrams are used to model low-level statistical techniques to analyse parts of the survey data. Technique diagrams are used to generate R scripts and web services providing remote access to the encapsulated scripts. Services can be orchestrated to realise the surveying.

Figure 1.3 shows an example of three SDL diagrams modelling a NZ Crime Victimization Survey (from [50]). After creating the overall survey structure a statistician creates SDL views to specify the survey process and techniques to use in detail. Figure 1.3 (a) shows a hierarchical task diagram, specifying two data analysis tasks to be carried out on the survey data. During data collection, the main concern is to specify sampling techniques used in the survey process and types of statistical metadata related to collected data. Figure 1.3 (b) specifies the two sampling methods to be used in the survey. Here the sampling frame is stratified in two stages by the modified area unit (1) then household visits planned using patterned clustering (2). Data Frame and Sample data icons indicate data used in pre-testing and collection phases. Figure 1.3 (c) shows a survey technique diagram describing the data analysis operations implementing the tasks in Figure 1.3 (a). Two visualisation methods (boxplot and multivariate analysis) are used to assess whether there is evidence of a statistical association between data variables. It generates textual R script implementations of aggregate techniques and uses R scripts to implement its low-level technique components. Figure 1.3 (d) shows links

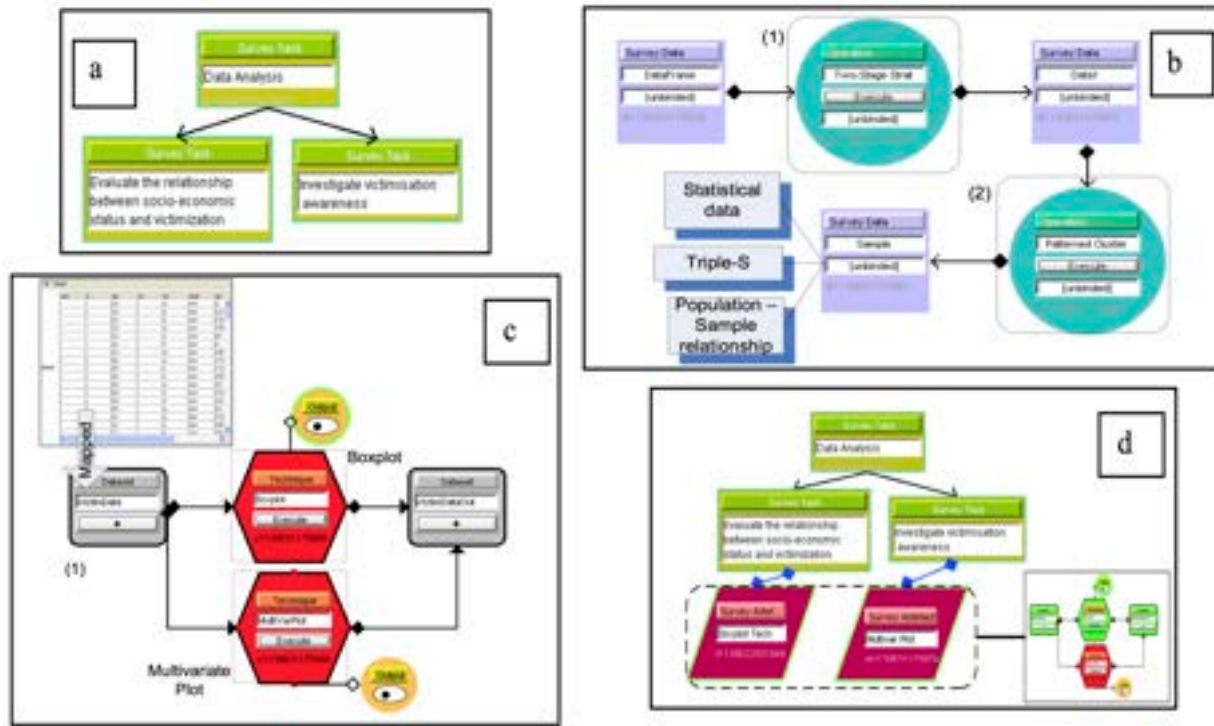


Figure 1.3: Statistical Design Language (SDL) being used to model (a) survey tasks; (b) survey sampling techniques; (c) an analytical technique; and (d) links between diagrams (from [50]).

between diagrams and artefacts.

SDL provides a powerful range of multi-level abstractions for statistical survey designers, statisticians being the target end users. This includes high level design, data specification, process flow, low-level statistical techniques, and a wide range of visualisations to show survey data analysis results. SDLTool also provides capabilities to generate reusable aggregate services that implement complex processes and techniques for reuse by other service-oriented applications. However, SDL can not express general purpose programming concepts and constructs and nor is it designed for general purpose business intelligence applications.

1.2.2 Why use DSVLS

Sometimes it is not obvious that a DVSL-based solution either exists or is necessarily a better solution than a textual Domain-Specific Language (DSL) or a general purpose modelling approach or programming language. Some indicators I have found useful when exploring the use of DSVLS for application to a new domain are outlined below. DSVLS are a useful modelling approach in situations where:

- a domain has a set of domain-specific concepts capturing rich properties about the domain that lend themselves to being captured in a meta-model;
- experts in the target domain have a set of notations or representations of the domain concepts they regularly use to describe aspects of the domain in designs, meetings, to explain models etc;
- models describing solutions in the domain can be readily constructed from these notations and underlying concepts far more readily and efficiently than if using more general purpose modelling languages and tools;
- domain-specific models can be used to synthesize general models e.g. code or parts of general models to realise solutions
- target end users find the domain specific modelling concepts, notations, tool support for authoring the models and support for generating solutions (or partial solutions) from the models more effective and efficient than using general-purpose modelling tools and approaches.

A good indication of a DSVL being potentially useful is when talking with software engineers or target end users – and/or visiting their work spaces or observing them interacting in meetings – a set of informal

DSVLs and their related domain-specific concepts are frequently used. Business process designers and indeed many business process improvement activities for many years have adopted BPMN and EML-style modelling constructs to design how the processes work, who enacts them, and related artefacts and data.

If a domain has a set of limited concepts that naturally provide an abstract description of a solution in that domain, this can be indicative of potential for DSVL (or DSL or combination) solution. These concepts are limited in scope i.e. are domain-specific, but provide powerful constructs for expressing complex ideas simply, consistent and elegantly. For example, the process stage, process flow, split flow, join, flow, actor and artefact concepts in business process modelling provide a very powerful, simple metaphor for describing a wide range of solutions. This leads to DSVLs like EML being able to model complex business process problems far more quickly and accurately and usefully than general-purpose approaches. The statistical surveying domain does not have an agreed DSVL or set of DSVLs, but we identified when talking with statisticians it does have well-defined concepts that we were able to map to SDL model concepts. From this we developed a set of DSVL elements to represent these domain-specific concepts and their inter-relationships.

While 3GL computer programming languages are very flexible and powerful, and it may be argued APIs and libraries provide sets of packaged abstractions for sub-domains, key disadvantages are the time it takes to construct solutions, repetitiveness, and lack of higher level abstractions than code constructs. DSVLs describing higher level domain elements can be enriched with properties to enable generation of whole or part 3GL programming language solutions. Enrichment may also be by relating one DSVL element to another e.g. a Horus-HPC parallel computation element in one view to a GPU grid compute element in another view, indicating how OpenCL kernel code is to be generated from the combined model.

1.2.3 Designing DSVLs

DSVL meta-models can be designed using conventional conceptual and data modelling techniques [51, 20]. Design concepts are identified from various information sources: domain experts; existing partial models (whether on paper or computer); existing 3GL or DSL solutions where meta-model elements and relationships are abstracted; or from databases, CSV and XML files, or other domain artefacts that capture part of the necessary modelling constructs for the domain. Often other solutions exist for the target domain, even DSVL based solutions, that provide most if not all of the information needed to define and construct the necessary meta-model for the new DSVL based solution [20].

For example, Horus-HPC's low-level models and parallel programming patterns are derived from the large body of work in this area over many years. However, its high level models we had to define after considerable work with scientists who develop their own bespoke HPC solutions for different domains [3]. EML uses BPMN and earlier business process modelling tool meta-model constructs to describe its overlay processes. SDL provides ways to package R scripts for reuse related to survey implementation, and structure data definitions and higher level survey process tasks and goals.

The most challenging – and often most creative – aspect of DSVL based solution design are the visual notations that provide much of the power and advantage of DSVL-based solutions [59]. A range of drivers influence how the visual notation is designed and how the overall solution will appear to target end users. These include but are not limited to:

- Who are the target end users of the DSVL? Are they familiar with visual oriented modelling approaches and support tools? Are they technically knowledgeable software engineers or non-technical domain experts or potentially both?
- What size and complexity of model will need to be represented?
- How big and complex will the resultant DSVL-based models get?
- if multiple DSVLs will be used to represent the problem domain, how do we link parts together across diagrams?
- What visual metaphors do the target end users work with now? Can we reuse and build on these in some way so the proposed DSVL-based solution will look familiar to them?
- Will DSVL models need to be shared among many people, live for a long time, be modified exten-

sively during their useful lifespan?

- Are there any existing DSVLs used for different problem domains that might translate well to the one under consideration? Can we learn from the successes (or failures) of these DSVLs in those other domains?

Practical considerations also need to be taken into account. Can the intended implementation platform for the DSVL tool actually support the range and complexity of the visual notations? Are the target end users going to be able to effectively and efficiently understand and use both the notations themselves and their editing tools? How do we handle challenges like version control, configuration management and collaborative editing of DSVL-based representations? Is the DSVL-based solution really better than using a conventional general purpose 3GL programming language (C, Java, Python etc), scripting language (R, Matlab, Perl etc). Is a special purpose DSL (textual) language more useful than a visual form?

These approaches are not always incompatible e.g. EMLTool supports both the EML DSVL and general purpose BPML; Horus-HPC includes a fully functional C code editor and extensive API library; and including components and scripts implemented in textual R code is supported by SDLTool.

Users of DSVL-based modelling solutions can be software developers e.g. HorusHPC, domain experts e.g. statisticians for SDL and business process experts for EML. Sometimes multiple user groups can be supported by the same tool/DSVL or subsets of the DSVL. Scientists can specify high level physical model principles in HorusHPC formula views and software developers translate these into detailed parallel code level DSVLs. Survey designers can specify goals and high level process tasks in SDL process diagrams, and expert statisticians and data scientists specify statistical technique details in Technique diagrams.

1.2.4 Realising DSVL-based Applications

Once a DSVL has been designed for a target domain we need to build a tool that supports the use of this DSVL. Given the complexity of such a task, a number of platforms have been developed to aid the creation of DSVL-based tools e.g. MetaEDIT+ [71], Eclipse GMF [24] and Microsoft DSL tools [16]. As can be seen from the representative DSVLs and their supporting tools in this chapter, a wide range of considerations need to be taken into account when building such a tool [63]:

- How large and complex with the DSVL tool likely be - how many DSVL diagram types, elements, and other features like code generation, data import or export, data visualisation etc? Different platforms provide different ranges of solutions.
- Can the tool platform handle the range of visual abstractions, appearances, composites and intended editing operations and interactions?
- Will complex data or code or scripts need to be imported? Generated and exported? Does the tool platform have the necessary capabilities?
- Will the tool interface need to be provided through a web browser or mobile phone?
- Are there approaches provided to supporting DSVL versioning, diffing, shared editing, large scale rendering etc?
- Will the DSVL-based tool need to be integrated closely with other tools / applications, and is there support for such integration in the DSVL tool platform?
- Can we impose various necessary constraints, checks and design critics on the DSVLs using the DSVL tool, in order to ensure models are correct, consistent and complete?

One of my main contributions over many years to making DSVL concepts realisable is the development of numerous supporting tools, both for the DSVL modelling and associated MDE-based support. These include but are not limited to MViews [38], JViews [27], JComposer [35], Pounamu [73], Marama [37], WikiBuilder [44], and Horus [3].

When developing prototype DSVLs and their support tools for research and practice, we want to evaluate both the support tool and its DSVL solution using a range of criteria [59, 23]. Learning from the results of these evaluations, we may want to refine the DSVL and/or its support tool to address issues our target end users have encountered and reported to us or that we have observed.

1.3 Model-Driven Engineering of Software

Doug Schmidt in a widely read introduction to a COMPUTER special issue on Model-driven Engineering describes MDE as:

“Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively.” [66]

Similarly to some of the drivers behind Domain-Specific Visual Languages choice in some domains, MDE approaches to software engineering attempt to address long standing issues including:

- textual programming languages (“3GLs”) are often too low-level to describe many abstractions in software engineering
- SE models are often too disconnected from 3GLs (program code) e.g. traditional analysis and design languages
- we often need high-level modelling languages to better express requirements, architectures, designs, tests etc through software engineering processes
- such SE models can be used to “construct” software directly i.e. translate from design-level elements to code-level elements via a “model transformation” approach – these models at various levels of abstraction can still be directly turned into/related to code constructs
- to do this we need to provide ways to build models, reason with models, translate models to(/from) code

Note that working with textual 3GL code is often still very useful/necessary even in systems where a lot of model-driven development support is used. Similarly, textual Domain-specific Languages (DSLs) can be used with visual DSVL models in domains where textual representations are more useful some of the time.

Many early MDE approaches originally focused on code generation. Increasingly, rather than generating just code from models, scripts or configurations or even other models are generated. For example, SDLTool generates R scripts from its survey technique diagrams, some of which may contain prepacked R script function calls themselves. EML generates BPEL4WS (Business Process Execution Language 4 Web Services) models that themselves can be run (enacted) via a workflow engine to orchestrate business process-implementing web services.

An example of such model-driven engineering from a textual Domain Specific Language is RAPPT [11]. Figure 1.4 outlines its process. RAPPT takes a textual design-level mobile App Description – sometimes called a Platform Independent Model (PIM) (it can also take a DSVL that represents limited, high-level parts of the textual app model). It has a model transformer that transforms this abstract design-level PIM App Description into a much more detailed code- and API-level Android Model for the Android platform mobile apps – sometimes called a Platform Specific Model (PSM). It then uses a further Code Generator component to translate the Android Model combined with a set of Code Templates into Android mobile app implementation artefacts – source code but also scripts, manifest, directories, XML and iconic elements. These can then be built by an Android development environment to implement the modelled mobile app. RAPPT allows generated Android code to be edited by the developer to add further features not supported by its code generated or to refine the skeleton mobile app implementation into a fully-fledged app product.

1.4 DSVLs and MDE for Software Engineering

Bringing these two approaches together – modelling complex software systems with DSVLs and using these DSVL-visualised models in MDE processes -- has been a key focus of a large part of my research work to date. Key requirements for such a combined approach, as outlined in Figure 1.5, include:

- a set of domain meta-model(s), model instances are used to model the problem domain;

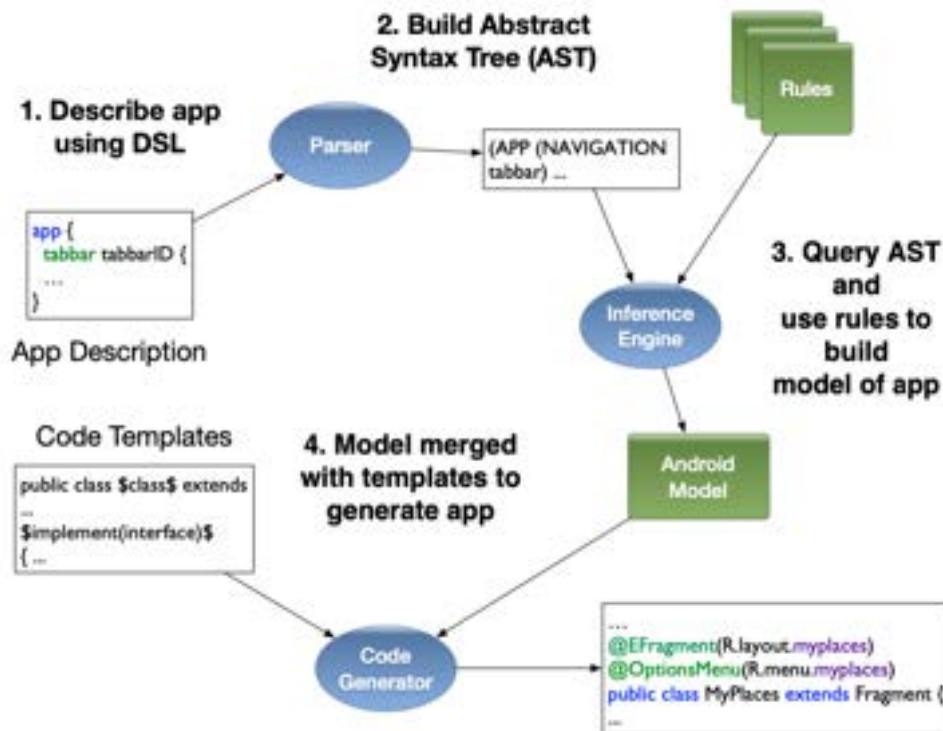


Figure 1.4: Example of MDE-based textual app model to android model to code generation process in RAPPT (from [11])

- a set of visualisation(s) of these domain model instances -- textual and graphical -- allow software engineers and/or end users to construct and refine problem domain solutions;
- mappings between models allow a tool to transform higher level models to lower level ones, including to code, scripts, XML etc.;
- a high level model might be transformed into a lower level model e.g. a requirements-level problem space model to a design-level solution space model, or a platform independent model enriched to become a more detailed platform specific model;
- low-level models are transformed to implementation-level code, scripts, configurations, etc that can be compiled to implement the system, or may be interpreted by an engine to realise the software desired;
- editing tools for DSVL-based model visualisations;
- transformation support i.e. model->model, model -> code transformers;
- visualisation support of models is sometimes useful e.g. code/data -> model -> DSVL representation;
- reasoning support e.g. analysis of models -- completeness, correctness, consistency; and
- model management support e.g. version control, diffing/merging, team collaboration support etc.

To illustrate the variety of ways this DSVL+MDE approach can be used, in the following subsections I illustrate a few example DSVL+MDE tools from my work. Some of these are targeted at supporting software engineers performing specific tasks. Some are targeted at supporting domain experts (non-technical end users). For each example I outline its problem domain; target user group(s); key meta-model elements; DSVL(s) used; transformation approach used; and target generated artefacts.

1.4.1 Performance Test-bed Generation Tools

I have led research into numerous performance engineering tools using DSVLs and MDE approaches. The initial ideas for this line of research came from my time in industry in the 1980s where I had to try and improve performance of complex database systems. This required writing many testing scripts – or “performance test-beds” – that was very tedious work. However, many of these test-bed scripts had great similarities and if a high level model of the target system structure could be defined, much if not all the performance testing scripts could be automatically generated from these models. Figure 1.6 shows one such tool in use,

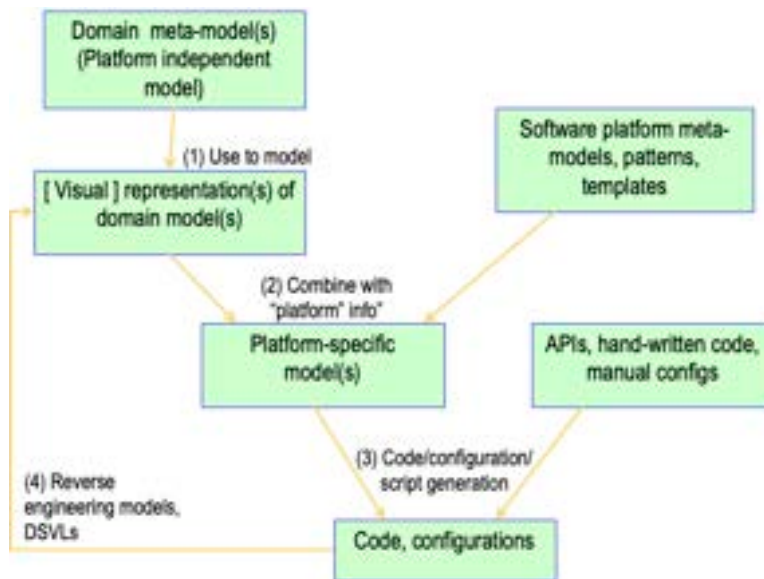


Figure 1.5: Outline of the DSVLs + MDE engineering process

(1) Graphical model editor showing a tree view and a central diagram.

(2) Graphical model editor showing a detailed diagram of the testbed.

(3) XML configuration for a test plan:

```

<TestPlan>
<hashTree>
<ThreadGroup guiClass="ThreadGroupGui" testClass="Th
enabled="true">
<stringProp name="ThreadGroup.name_time">1</stringProp>
<boolProp name="ThreadGroup.scheduler">false</boolProp>
<stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
<longProp name="ThreadGroup.start_time">5576438592000</longProp>
<elementProp name="ThreadGroup.main_controller" elementType="LoopController"
guiClass="LoopControlPanel" testClass="LoopController" testName="Loop Controller" enabled="true">
<boolProp name="LoopController.continue_forever">false</boolProp>
<stringProp name="LoopController.loops">1</stringProp>
</elementProp>
<stringProp name="ThreadGroup.num_threads">5</stringProp>
<stringProp name="ThreadGroup.duration"></stringProp>
<stringProp name="ThreadGroup.delay"></stringProp>
<longProp name="ThreadGroup.end_time">1076438592000</longProp>
</ThreadGroup>
</hashTree>

```

(4) Java code for a client thread:

```

class ClientThread extends Thread
{
    static Object critsect = new Object();
    public synchronized void default() throws Exception
    {
        while(isAlive())
        {
            wait(5);
        }
    }
    private static Random random = new Random();
    public static synchronized int getRandom(int n)
    {
        return random.nextInt(n);
    }
    // generate code for each page in the Pagefile specification
    private String page_index() {
        synchronized(critsect) {
            int page_index =
        }
        // wait for specified amount of time
        try{
            sleep(getRandom(50)+500);
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

Figure 1.6: Example of the MaramaMTE performance testbed generator in use (from [19])

MaramaMTE [19].

MaramaMTE has the following key features:

- Problem domain: generation of performance test bed code/scripts from high level modelling of software architectures of distributed systems
- Target user group(s): performance engineers (typically highly experienced software engineers)
- Key meta-model elements: architectural elements of distributed systems – clients, servers, databases, network connections, compute and data applications ; loading models for clients e.g. number users, number transactions per second, types of transactions etc
- DSVL(s) used: (1) architecture model with detailed characteristics of each element (representing key software components); and (2) stochastic form charts (representing probabilistic client loading models)
- Transformation approach used: XSLT scripts and Java code
- Target generated artefacts: client, server code, loading scripts, build scripts, deployment scripts

When using MaramaMTE, the performance engineer models their system’s software architecture using architectural view DSVLs, as shown in Figure 1.6 (1). They model client loading models using stochastic form charts, as shown in Figure 1.6 (2). The XSLT-based code generators synthesize from these two DSVL models scripts e.g. for Apache JRunner (3) and Java client and server code e.g. Figure 1.6 (4). MaramaMTE generates real code that is compiled and run for clients and servers. The loading scripts have the clients run a very large number of transactions against the servers and MaramaMTE collects results for presentation to the performance engineers.

1.4.2 Data Transformation Tools

Integrating complex distributed systems by exchanging complex data is a very common need in many systems. Originally Electronic Data Interchange messages were used and the writing of EDI encoding and decoding software to support EDI message exchange between systems is a very challenging task. We developed a tool with Orion Health in a collaborative R&D project to facilitate the modelling of complex EDI messages for health system data exchange, the Orion Message Mapper [34], eventually commercialised as the Rhapsody message mapping engine suite. This approach used hierarchical tree-based message mappings to specify correspondences between source EDI message elements and target EDI message elements, and formulae to translate source data to target data formats. While suitable for software engineers to use, the original target users of Orion Message Mapper, they are not very suitable for non-technical domain experts.

This led to a new approach I invented, and supervised a Masters student to prototype and evaluate, the Form-based Mapper [53]. This uses form visualisations of XML data models – meant to be analogous to the business forms traditionally used to exchange data between businesses e.g. fill out an order form, send to supplier, supplier copies data from form to another format, supplier processes order etc. Figure 1.7 shows an example of the Form-based Mapper in use.

Key features of the Form-based Mapper include:

- Problem domain: data exchange between complex business systems
- Target user group(s): non-technical domain experts
- Key meta-model elements: business form structures – represented as XML models
- DSVL(s) used: hierarchical “form” visualisation meant to resemble real-world paper and electronic forms
- Transformation approach used: XSLT scripts
- Target generated artefacts: XSLT scripts

The domain end user e.g. someone who is knowledgeable about the data to be exchanged between businesses, imports an XML schema and has it visualised as a form like layout, as in Figure 1.7 (1). The user may rearrange the format to look more like a real-world paper or electronic form. They then specify via drag and drop “correspondences” between form elements, as shown in Figure 1.7 (1) and (2). Some of these, like the correspondences in Figure 1.7 (1) can be simple one to one mappings with little or no data format

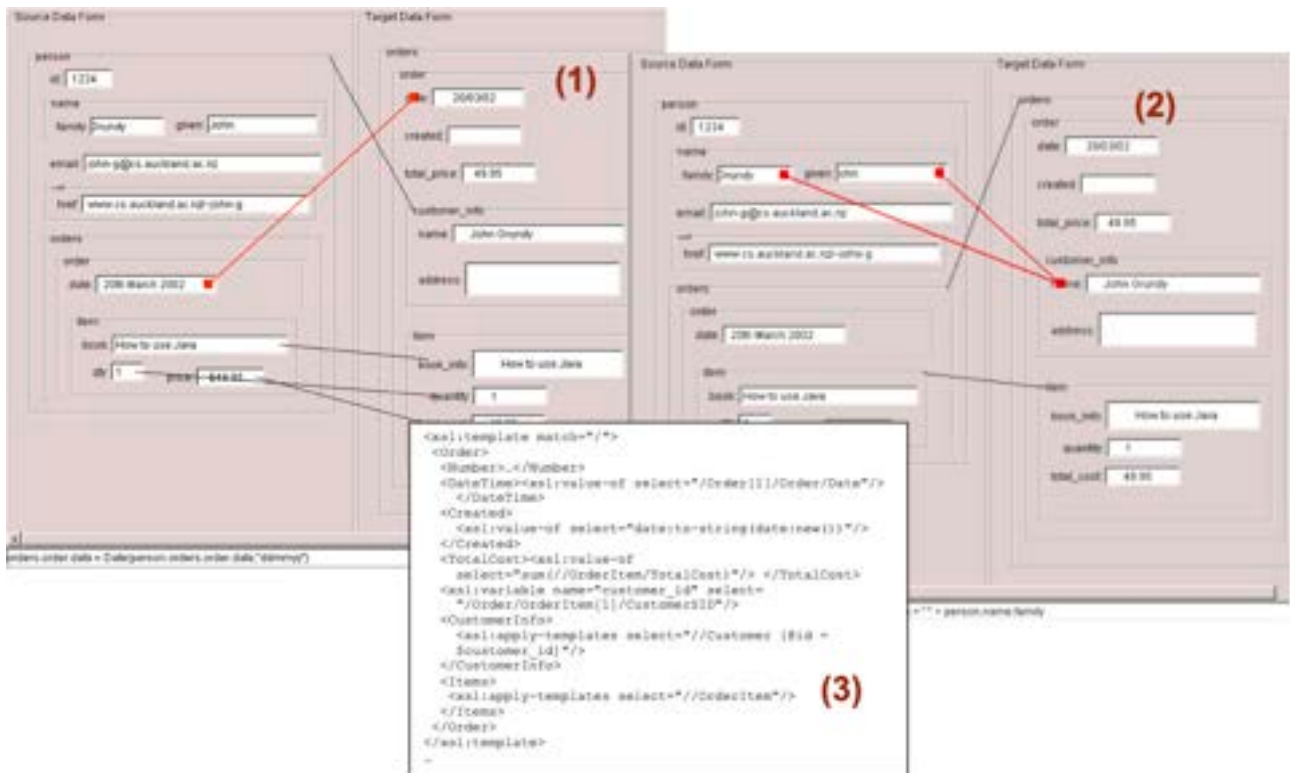


Figure 1.7: Example of the Form-based mapping generator in use (from [53])

translation; some can be 1:many or many:many complex data transformations. A formulae builder is provided (examples shown bottom text field in Figure 1.7 (1) and (2)) allows specification of formula-based data transformations, meant to be like spreadsheet-like formulae. The Form-based Mapper then generates XSLT scripts to implement XML to XML data transformations based on the specified mappings, part of one shown in Figure 1.7 (3). One issue we found when evaluating the format translation formulae is these are difficult to end users to use, being themselves based on XSLT-based formulae.

1.4.3 Mobile App Generation Tools

Developing mobile apps has become very popular but is still predominantly limited to those with high development expertise. Despite the availability of a range of low-code/no-code MDE-based app generation and configuration tools, these have major limitations around flexibility, expressive power, and quality of generated mobile app [10]. Despite these limitations of MDE-based app generation approaches, eHealth apps are a promising area for modelling and generating fully functional apps. This is because eHealth apps in specific domains share many commonalities. We wanted to support public health clinicians in modelling and generating eHealth apps for chronic disease management e.g. diabetes, obesity, etc. These all use a similarly-structured “care plan” concept and mobile apps provide patients with self-management steps following a set of exercise, diet, pharmacology and monitoring interventions. We developed a clinician-oriented chronic disease management app modeller and generator for this domain [48]. An example of this in use is shown in Figure 1.8.

Key features of this Visual Care Plan Modeling Language (VCPML)-based eHealth app generator include:

- Problem domain: eHealth apps for chronic disease management
- Target user group(s): clinicians model app care plans and tailor to individual patient needs, patients use generated eHealth app
- Key meta-model elements: care plans and app interface components
- Visual Care Plan Modelling Language (VCPML) and a visual interface specification language

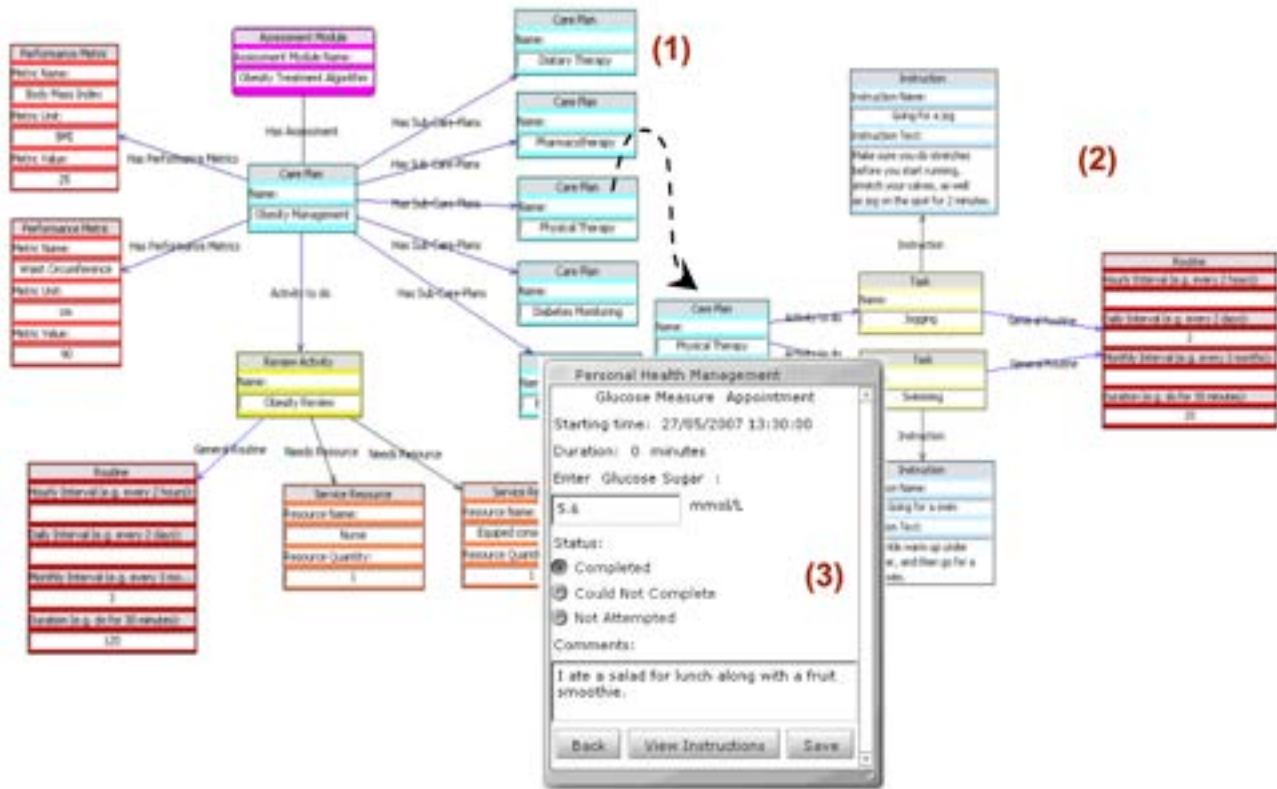


Figure 1.8: Example of the Visual Care Plan Modelling Language eHealth app generator in use (from [48])

- Transformation approach used: C implemented code generators
- Target generated artefacts: OpenLaszlo code which itself is then translated into Flash or Microsoft Mobile code to run on (old) phone platforms

Figure 1.8 (1) shows a VCPML diagram to support patients with obesity management. The idea for VCPML came from medical texts where these care plans were textually described, and we developed a meta-model and visual model to describe them. These can be hierarchical, with Figure 1.8 (2) showing a more detailed care plan relating to physical therapy after clicking on the icon in Figure 1.8(1). A further visual language (not shown) is used to describe the details of how to present care plan elements in a mobile phone GUI. An OpenLaszlo implementation of the mobile app is generated, itself then transformed into one or more specific mobile app implementations. The one shown in Figure 1.8 (3) is a Flash-based implementation running on a handset with a Flash player embedded. This is a 1000% code generation approach - unlike our RAPPT tool described previously, the generated code can not be changed. While in theory the VCPML approach to chronic disease management app generation was good, a number of limitations were encountered by users. These included confusing user interface specification language; lack of accounting for diverse users of the mobile apps in the generated apps – a one size fits all approach; and inability to change or augment generated app appearance and functionality.

1.4.4 DSVL Tool Generators – “DSVL Meta-tools”

As discussed previously, we need to implement sophisticated support tools in order to realise DSVL and MDE based approaches. We have found that such DSVL-based tools themselves are amenable to using DSVL+MDE approaches, given they have many commonalities, common meta-models and DSVLs can be developed to describe them, and much of their functionality can be generated, as either code or configurations. I have used DSVL and MDE techniques to specify and generate many DSVL-based tools - I call these meta-tools or meta-DSVL tools [35, 37]. One such example is the VikiBuilder, a DSVL and MDE-based tool for specifying Visual Wikis [44]. An example of VikiBuilder in use to specify a “Lost” TV series Visual

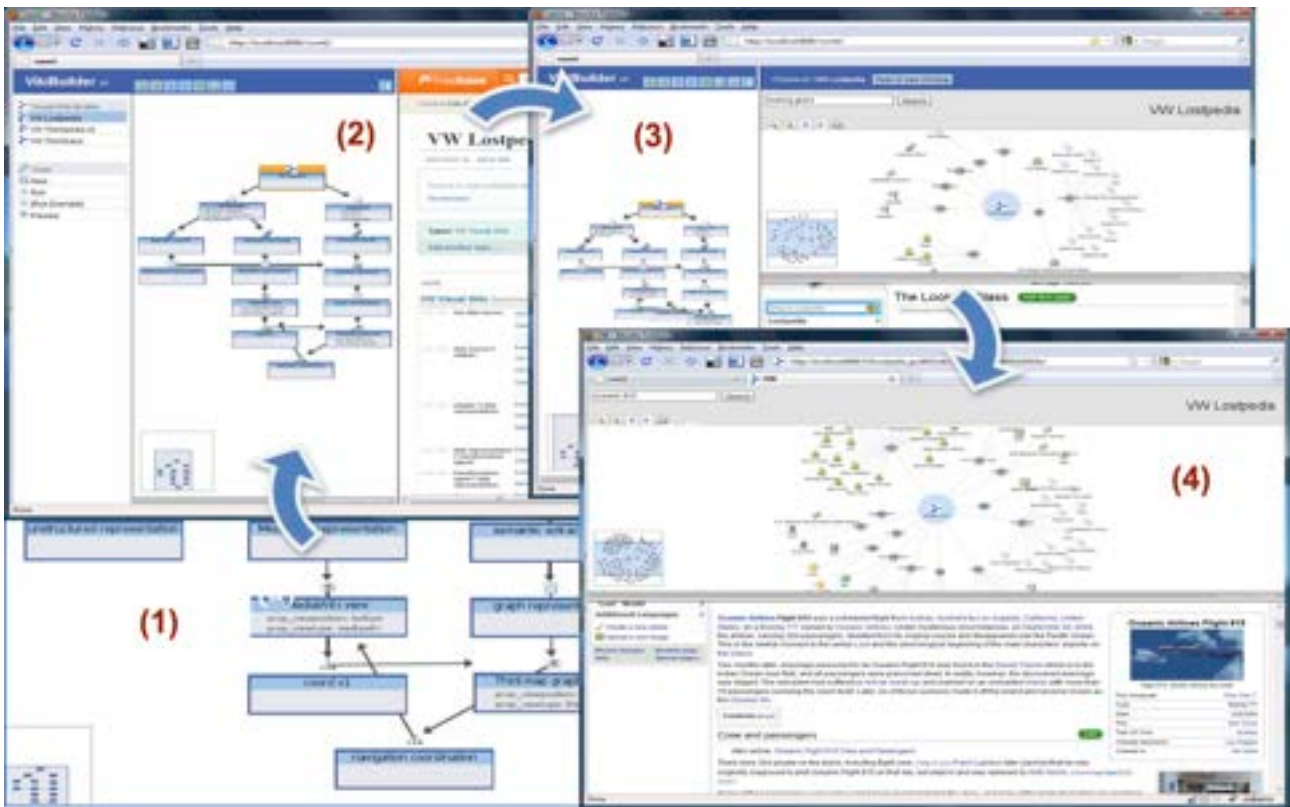


Figure 1.9: Example of the VikiBuilder Visual Wiki designer and generator in use (from [44])

Wiki is shown in Figure 1.9.

- Problem domain: Generating “visual wikis”
- Target user group(s): Visual wiki designers - not necessarily software engineers (unless they want to author their own Java plug-ins to extend the platform – see below).
- Key meta-model elements: key visual wiki elements e.g. data sources, data filters, data transformations, various visual wiki screen elements
- DSVL(s) used: a simple box and line visual wiki element composer, itself realised as a Visual Wiki (thus VikiBuilder is itself a “meta-Visual Wiki”)
- Transformation approach used: Java code creating database content
- Target generated artefacts: creates a database containing configuration information for the new visual wiki, which is interpreted to create the new Visual Wiki on the Confluence platform ; additional hand-implemented Java code plug-ins can be added to enhance functionality

Figure 1.9 (1) shows part of the specification of a new Lostpedia Visual Wiki that is intended to provide an interactive, visual way of exploring the Lostpedia site. Various details about data sources, filters, transformations, aggregations, and visualisations are specified using form-based information associated with each visual wiki DSVL element, as shown Figure 1.9 (2). Note the VikiBuilder tool is itself a Visual Wiki built on top of the Confluence Enterprise Wiki platform with a set of Java plug-ins. The new visual Wiki can be tested and specifications, both DSVL elements and their properties updated interactively, shown in Figure 1.9 (3). Finally the complete Lostpedia Visual Wiki can be deployed for use, shown in Figure 1.9 (4).

The generator takes the specified visual wiki information and populates a database with essentially a set of detailed configuration model information, rather than generating code to implement the Visual Wiki. A single Visual Wiki code platform thus interprets different detailed specification models to produce quite different Visual Wikis. The VikiBuilder is thus a good example of model to model transformation using MDE, rather than model to code. In addition, software engineers can specify Java-based plugins to include in the new visual wiki that use a set of APIs to extend the platform capabilities, to include functionality not built into our original Visual Wiki platform. The VikiBuilder also thus illustrates semi-automated use of

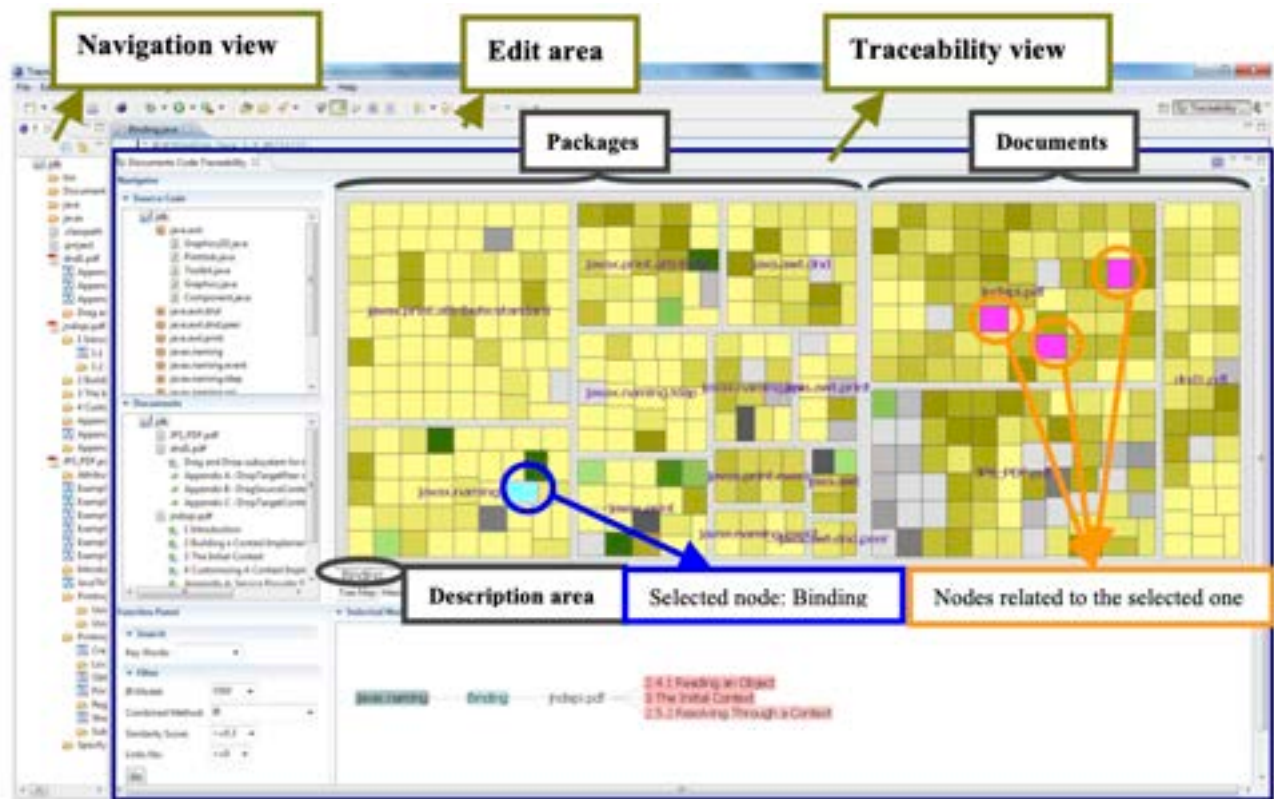


Figure 1.10: Example of the DCTracVis traceability link reverse engineering tool in use (from [15])

MDE to produce solutions – the DSVL allows modelling of a wide range of concepts, but ones not catered for it supports specifying Java plugins to use that are hand-coded by developers.

1.4.5 Reverse-engineering DSVL Models from Code

All of the examples I have shown so far of DSVLs and MDE use “forward engineering” – high level model to lower level model to code/script/configuration model. Sometimes it is useful to use “reverse engineering” where a high level DSVL-visualised model is extracted from lower level models, code, documentation, performance logs, etc. An example of such a tool is DCTracVis, a tool that we developed to address the problem of visualising a large number of reverse-engineered traceability links between code and documentation [15]. Figure 1.10 shows an example of DCTracVis in use.

Key features of DCTracVis include:

- Problem domain: want to visualise and explore interactively a large number of reverse-engineered traceability links between source code and documentation
- Target user group(s): software engineers
- Key meta-model elements: code abstract syntax tree, document paragraphs and words, and links between code tree elements and document words/phrases.
- DSVL(s) used: a heat map and tree visualisation
- Transformation approach used: Java-based reverse engineering tool to extract models from Java source code and PDF documents
- Target generated artefacts: higher level model of trancelinks between code elements and documentation elements

Figure 1.10 shows a complex Java program that has been analysed and its Packages (collections of Java classes) shown as group of heat map visualised items (left hand side “Packages” heat map). A set of PDF documents and section headings within the PDFs are show in the right hand side heat map (“Documents”). When the software engineer clicks on a node – a Java class – in the left hand side Packages heat map, elements

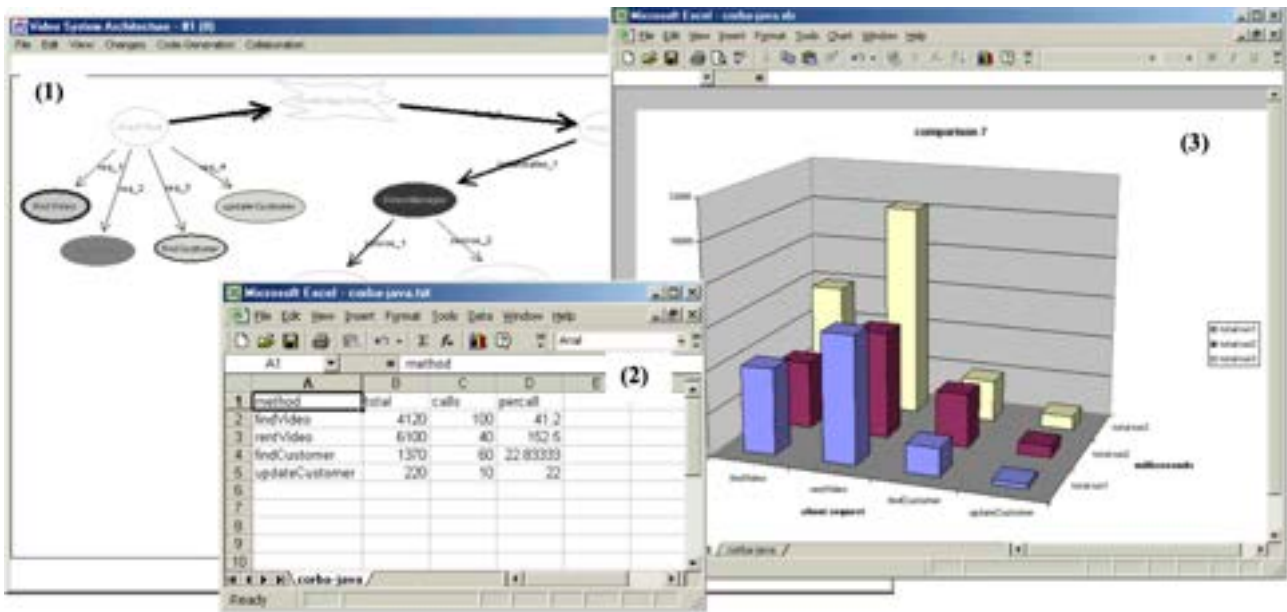


Figure 1.11: Example of the SoftArch/MTE performance engineering tool showing augmented DSVL with reverse engineered data (from [26])

corresponding to sections in the documentation about this class are highlighted in the right hand side Documents heat map. A tree visualisation of links is also shown at the bottom, here showing a traceability link from the selected javax.naming class to three sections in the jndspi.pdf documentation file explaining uses of this Java class. The developer can then go into the PDF at each indicated section to see potentially useful information about using this javax.naming class.

Tools that use DSVLs and MDE for forward-engineering can also use reverse-engineering and annotation of their DSVL diagrams to show e.g. run-time reverse engineered information. An example of this is used in our SoftArch/MTE, Marama/MTE and Cloud/MTE performance engineering tools [26, 19]. These extract low-level run-time performance data after running performance tests, abstract this data into high level performance summaries, and visualise these summaries by highlighting DSVL diagram icons. Figure 1.11 shows an example of this in SoftArch/MTE [26]. A software architecture DSVL model (1) is highlighted with shading and line thickness to show – at a high architectural level – places in the architecture design causing potential performance bottle-necks. The data is abstracted from run-time captured low-level performance data, shown in (2), and can also be visualised in an alternative way using a bar chart, as shown in (3).

1.5 Overview of the papers in this Thesis

Figure 1.12 shows a summary of many of the tools and approaches described in the papers that make up this thesis, with an indicator to which part/chapter they appear in. Each tool/approach is briefly introduced and summarised in the following subsections. At the top of Figure 1.12 are several DSVL-based meta-tools and various extensions to support more human-centric and collaborative modelling with these DSVLs. A range of software engineer-supporting tools are shown below the timeline in Figure 1.12 in blue, supporting a range of requirements, architecture, design, coding, testing and process management tasks. A set of end-user oriented tools are shown in red at the bottom of Figure 1.12.

The collection of papers I include in Part 1 describe ways in which DSVL-based MDE tools can be described and implemented, including themselves using DSVL and MDE-based approaches. In Parts 2-4, a collection of papers describe why DSVLs can be a good choice for various aspects of software engineering, including requirements engineering, software architecture, design, testing and to support software process modelling and enactment. In Part 5, the selected papers describe a body of my work creating various “human-

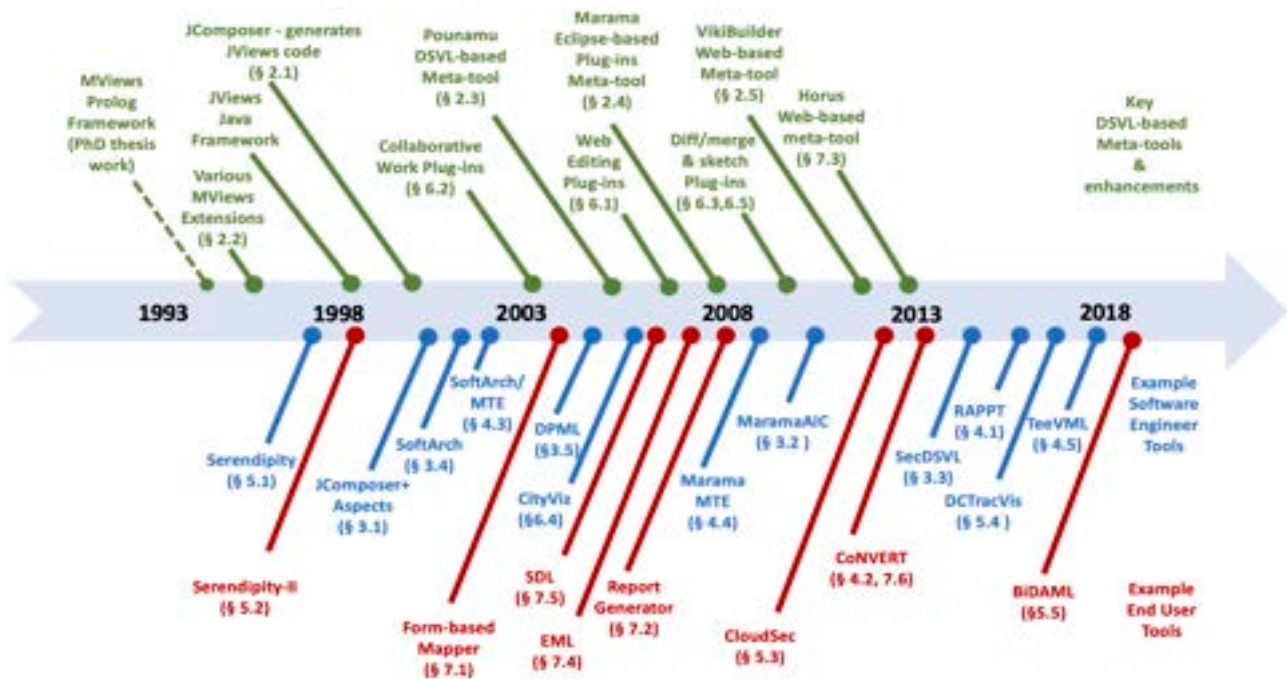


Figure 1.12: Approximate timeline of this thesis collected contributions

centric” and collaborative support facilities that can aid the use of DSL-based MDE approaches in various ways. In Part 6 I include several papers presenting reasons and examples of why DSL-based MDE tools can also be used to support non-technical software end users to model and build their own software, in a wide variety of constrained domains. Finally, in Part 7 a recent paper outlines some future directions in supporting more human-centric model-driven software engineering using DSLs.

1.5.1 Part I – Modelling tools and their development

In order to realise the approach of DSLs and MDE to produce software systems, we need tools to model with DSLs and generate other models/code/configurations etc. from these models with these DSLs. The papers in this part of the thesis describe selected examples of a range of solutions we have produced to realise such tools. These examples include post-PhD work on a number MViews framework extensions, the JViews framework, the JComposer DSL- and MDE-based JViews modeller and code generator, the Pounamu DSL-based meta-tool, the Marama Eclipse IDE-based meta-tool, and the VikiBuilder Visual Wiki modeller and generator. Using these platforms my collaborators and I have realised dozens of innovative DSL and MDE-based tools for both software engineers and end users in diverse application domains. These results have been published in well over 200 of my papers.

MViews (implemented in an OO Prolog) and JViews (implemented in Java) are Object-oriented (OO) frameworks for building DSL-based tools, and provide some code and model generation support using MDE. However building DSL-based tools with these frameworks requires textual-coding and specialising complex framework classes, time-consuming and only suitable for expert programmers to do. *“Constructing component-based software engineering environments: issues and experiences”* [35] describes JComposer, itself a JViews-based DSL and MDE meta-tool for modelling and partially generating JViews-based DSL and MDE tools. We describe the need for DSL-based meta-tools like JComposer, its capabilities and evaluation of its support, including architectural support for distributed, collaborative work specifying and partially generating DSL-based tools. JComposer supports modelling of partial DSL tools. It then uses MDE to generate partial DSL tool implementation code using the JViews Java class framework. These partial tool implementations are then completed by tool developers modifying and extending by hand the generated Java class code to complete the tool. JComposer results in much quicker/easier DSL-based tool development in JViews than using JViews alone. JComposer, like JViews, was intended for software

engineers to use, as it requires quite a lot of coding knowledge for its Java framework class specialisation and implementation.

Inconsistency between DSVL views frequently occurs when a designer modifies e.g. a high level process model or requirements model using a DSVL, but it is unclear how this change can/should be translated to a design-level DSVL model or code-level text. This becomes a more complex problem as more diagram (view) types are added, collaborative work between multiple designers is supported, and various kinds of models are integrated. *“Inconsistency Management for Multi-view Software Development Environments”* [32] describes a range of extensions made to the Prolog-based MViews platform (originally developed in my PhD) and its successor Java-based JViews framework to support inconsistency management in DSVL-based tools. We show a range of inconsistencies that can result and novel ways to manage them when trying to keep various general-purpose and domain-specific graphical representations of software engineering models consistent. MViews and JViews were both intended for software engineers to use, as they require a lot of coding knowledge for their OO framework class specialisation and implementation.

Pounamu: a meta-tool for exploratory domain-specific visual language tool development [73] describes Pounamu, the successor DSVL meta-tool to JComposer/JViews. Pounamu is a stand-alone, Java-implemented meta-tool with a greater range of DSVL-based meta-tools, and it generates DSVL tool specification files which are interpreted by Pounamu itself to realise the target DSVL-based tool. Pounamu supports development of Java code-based plug-ins to extend the generated tool functionality in more seamless ways than JViews. A number of extensions to Pounamu support collaborative work and web- and mobile-based editing, described in later thesis chapters. We have used Pounamu to build a wide range of DSVL-based tools for software engineers and end users. Pounamu was intended for software engineers to use, as it requires some coding knowledge for its Java plug-ins, though we have also had some end users successfully use it to develop their own DSVL-based tools in limited ways.

Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications [37] describes Marama, the successor to Pounamu. Marama is a DSVL meta-tool realised by a set of Eclipse IDE-based plug-ins, rather than a stand-alone tool like Pounamu. Marama provides much more sophisticated DSVL-based tool specification than Pounamu or JComposer, and many more extensions via its own Eclipse plug-ins and also via many third-party Eclipse IDE plug-ins. This includes Eclipse-based code development IDEs, making Marama-based tools much more closely integrated with other development tools than in our previous meta-tool approaches. Marama even has a DSVL-based MDE code generator specification and generation tool – a DSVL-based meta-MDE tool [45]. It also has a range of novel DSVL-based design critic and constraint modelling and generation tools, these making complex DSVL-based tool functionality much easier to build than Pounamu and JViews/JComposer [2]. We have used Marama to build a wide range of DSVL-based tools for software engineers and end users. Marama was intended for software engineers to use, but we have had non-technical end users successfully use it to build basic DSVL-based tools.

Finally in this part I describe *VikiBuilder: end-user specification and generation of Visual Wikis* [44]. As previously summarised, VikiBuilder is a web-based, DSVL-based Visual Wiki specification and generation tool. It is itself a Visual Wiki and generates Visual Wiki configuration data via MDE from its DSVL-based models. Unlike the previous meta-tools, VikiBuilder was always intended for end user, non-technical user specification and generation of Visual Wikis. It can also be used by software engineers who can develop Java-based plug-ins to extend the Visual Wikis generated by VikiBuilder.

1.5.2 Part 2 – Requirements and Design support with DSVLs and MDE

We have used our meta-tools from the previous section, and others, to develop a wide range of tools to support software engineers during requirements engineering and software architecting tasks for complex software systems. An early example is a set of extensions to existing DSVLs – those of JComposer – described in *Aspect-oriented Requirements Engineering for Component-based Software Systems* [25]. In this work, I invented a set of novel requirements-level “aspect” annotations on high level components describing cross-cutting problem space concerns. These augmented JComposer requirements-level specifications and allow a

software engineer to describe and reason about cross-cutting concerns in DSVL-based tools, as well as other DSVL-described software requirements. We later added this concept to general purpose UML diagrams, used them to augment design level models, and used them to describe implemented software component capabilities to support run-time dynamic integration of component-based systems. JComposer uses MDE to take its aspect-augmented requirements DSVL models and generate design-level detailed aspect information (via model to model transformation).

MaramaAIC: Tool Support for Consistency Management and Validation of Requirements [46] describes MaramaAIC, a tool supporting requirements engineering using Essential Use Case (EUC)-based models and Essential User Interface (EUI) models. MaramaAIC provides a novel DSVL-based representation of these EUC based requirements models and uses MDE to generate EUCs from essential interaction models extracted from natural language text. It also generates example user interface mockups using MDE from its EUI based DSVL models. MaramaAIC was, as the name suggests, implemented using our Marama meta-tools.

Software security engineering is challenging. *Adaptable, Model-driven Security Engineering for SaaS Cloud-based Applications* [5] describes several DSVLs to support modelling different aspects of software security, including requirements-level and design-level characteristics. MDE-based tool support enables requirements level software security properties to be translated to solution space architecture and design level choices to realise security requirements. Further MDE support assists developers in encoding these DSVL-specified security solution decisions into software component code and configurations. This includes supporting run-time security property management. Finally, low-level run-time security monitoring data can be reverse-engineered and abstracted into design level DSVL-visualised information for system security managers.

SoftArch is a tool I developed to model a range of complex software architecture abstractions. **SoftArch: tool support for integrated software architecture development** [29] describes the DSVLs provided by SoftArch to support a range of software architecture modelling at various levels of abstraction. MDE techniques are used to generate partial OO design models to exchange with other modelling tools. Reverse engineering is used to provide dynamic visualisation of running systems based on these models. SoftArch DSVLs can be augmented by running system data to debug and understand how the systems work.

Design patterns are reusable solution approaches to tackling common design and programming problems. They are often described with UML-based design models. *A Visual Language for Design Pattern Modelling and Instantiation* [56] describes a novel DSVL, the Design Pattern Modelling Language (DPML), used for describing design patterns, and MDE techniques for realising these in programming code. A supporting tool, DPMLTool, was originally implemented with the JViews/JComposer meta-tool. A more advanced version, MaramaDPML, was subsequently reimplemented with the Marama meta-tool.

1.5.3 Part 3 – Development and Testing with DSVLs and MDE

We have invented many innovative DSVL- and MDE-based tools to support design, implementation and testing of software systems. *Supporting Multi-View Development for Mobile Applications* [10] describes RAPPT, a DSVL- and DSL-based mobile app code generation tool. Described previously, RAPPT supports combined visual DSVLs for high level mobile app modelling combined with more detailed textual DSL models to describe lower level app designs. These are then used to generate a fully functioning Android app including code, manifest, configuration and build files. The generated code is intended to be further hand-edited to polish and complete the app. RAPPT was intended for professional app developers to increase their productivity.

I have described many uses of model transformation, used in our DSVL meta-tools and in several example tools. Most of these are usually implemented as Java code, XSLT transformation scripts, or using other textual DSL code generator scripting languages. *Specifying Model Transformations by Direct Manipulation using Concrete Visual Notations and Interactive Recommendations* [7] describes CoNVERT, a DSVL-based model-to-model and model-to-visualisation mapping and generation tool. This paper de-

scribes the visual model-to-model specification aspects of CoNVERT – its domain-specific DSLs used to visualise XML models, its model to model visual mapping specification DSL, and its mapping generator that uses MDE to transform its models to a detailed XML model transformation implementation. CoNVERT was intended for non-technical end users, but can be used by software engineers too.

SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions [26] describes extensions to SoftArch including augmented software architecture DSL diagrams and MDE to generate performance test-beds for these architectures to support large scale software performance engineering. As discussed previously, testing whether a planned complex software system will meet its performance targets is very challenging. In this paper we describe how we augmented the original SoftArch DSL-based software architecture models to add detailed implementation platform and client, server and database properties and templates to synthesize realistic client loading models and server and database models. We describe how we generate these detailed models using XSLT-based model transformation approaches. We also describe the reverse engineering and abstraction of detailed run-time performance data into summaries and display of these to the user by augmenting the architectural DSL models. SoftArch/MTE was implemented with our JViews/JComposer meta-tool.

Previously I described and illustrated MaramaMTE, a successor to SoftArch/MTE for distributed system test bed generation and performance engineering. *Realistic Load Testing of Web Applications* [19] describes using novel DSL-based models in MaramaMTE – augmented architecture diagrams and augmented stochastic form charts – to model and generate via MDE web-based system performance test beds. Unlike the JViews-based SoftArch/MTE, MaramaMTE is fully integrated into the Eclipse IDE as a toolset and makes use of third party Eclipse plug-ins to provide a much more integrated performance engineering toolset for software engineers.

Finally in this part I describe TeeVML, a performance emulation *environment* generator. SoftArch/MTE and MaramaMTE described above generate fully functional models of complex distributed systems that are compiled and run with associated environment software (database servers, web servers etc) to performance test them. In contrast, TeeVML and its supporting tool generate “emulation environments” to test the behaviour and performance of real, very large scale software systems in mock emulation environments – essentially the opposite of MaramaMTE and SoftArch/MTE. *A Domain-Specific Visual Modeling Language for Testing Environment Emulation* [54] describes this approach where a new DSL-based high-level model of “system endpoints” – basically models of complex software system interfaces – are used to model and generate the complex environment a real-world software system would have to operate in. This software system is then run in this generated emulation environment, instead of having to hand-construct a (very) complex testing environment for it. TeeVML is implemented with the commercial MetaEDIT+ meta-tool.

1.5.4 Part 4 – Process and Project Management with DSLs and MDE

Software process models can be very complex. In the 1990s there was a lot of interest in (semi-)automated tools to model and enact (run) process models to guide software development. *Serendipity: integrated environment support for process modelling, enactment and work coordination* [39] describes Serendipity, such a process-centred environment. A set of DSLs are used by developers to model complex software processes. Serendipity then uses MDE to generate detailed software process models from these DSLs that are then run – or enacted – to implement the specified software process and guide developers in following it. Serendipity provides a range of high-level and detailed software process descriptions, including “agents” that monitor software development tool activities to semi-automate complex software processes. Serendipity users are software engineers, particularly project leaders. Serendipity was implemented with the MViews meta-tool framework.

A decentralized architecture for software process modeling and enactment [36] describes the successor to Serendipity, Serendipity-II. This supports a more powerful visual editing tool, more advanced collaborative work and third party tool integration, and some improved DSLs. Serendipity-II generates process models using MDE like Serendipity, but also generates Java code using MDE to implement a variety of tool

integration support features. End users of Serendipity-II were originally intended to be software engineers, but it can also be used by non-technical project managers in other domains. Serendipity-II was implemented with the JComposer meta-tool and JViews framework.

Specifying complex software security requirements and behaviours is challenging, especially for end users whose security requirements may evolve over time. In *Collaboration-Based Cloud Computing Security Management Framework* [4] we present a combined DSVL- and DSL-based framework to support the process of specifying and enforcing end-to-end, complex, cloud-based Software as a Service (SaaS)-based application security requirements. From these models we generate detailed security properties for the target system using MDE. These models are then used at run-time to configure a running cloud-based SaaS application's run-time security enforcement approaches. End users are SaaS application owners, often non-technical end users. Software engineers can also use the toolset to model security requirements and have them enforced at run-time. An early version of our web-based Horus meta-tool [6] and a set of web forms are used to specify the security models.

DCTracVis: a system retrieving and visualizing traceability links between source code and documentation [15] describes the DCTracVis reverse engineering tool introduced previously. Unlike many of the systems presented in this thesis, DCTracVis abstracts higher-level models from low-level code – documentation links, reverse engineered using text processing algorithms. It then uses Heat Map- and Tree-based DSVL representations of these models to allow software engineers to browse between high-level links between documentation and code elements.

BiDaML is a tool for modelling complex data analytics applications, described in *An End-to-End Model-based Approach to Support Big Data Analytics Development* [47]. BiDaML uses several DSVLs ranging from high-level brainstorming diagrams to low-level technique diagrams to specify complex data analytics applications from varying perspectives. MDE-based generators produce detailed reports to guide data analytics teams, and partial Java and Python implementations of data analytics applications. End users are multi-disciplinary data analytics team members – domain experts, business analysts, software engineers, project managers and cloud platform experts. BiDaML is implemented with the commercial MetaEDIT+ meta-tool.

1.5.5 Part 5 – Human-centric DSVL Modelling and Collaboration

In this part of the thesis I describe several research works that look to support more “human-centric” modelling with DSVL-based MDE tools. By this I mean approaches to make the tools easier to use, support more “natural” modelling, and support multiple developers or end users working collaboratively together on models.

Experiences developing architectures for realising thin-client diagram editing tools [31] describes a variety of Pounamu extensions to enable users of Pounamu tools to edit DSVL diagrams using a web browser, mobile phone and even a 3D browser plug-in. The idea is that Pounamu is a Java-based application that needs to be installed on each users machine and regularly updated. Similarly, due to Pounamu's use of Java code plug-ins for much DSVL-based tool implementation, this makes sharing new DSVL tools and developing them collaboratively very challenging. Pounamu/Thin – the web/mobile-supporting extensions – instead host a single Pounamu instance on a server and provide web browser/mobile phone editing interfaces. Some of the web browser editors are quite sophisticated, using SVG and ECMA script to provide highly interactive diagramming very similar to the desktop Pounamu. Note also that this work was done in the mid-2000s, long before today's more powerful browser-based client capabilities were developed. Pounamu/Thin even allows Pounamu meta-tool specifications to be edited and thus new Pounamu-based DSVL tools to be designed using the browser-based interface.

In *Engineering plug-in software components to support collaborative work* [28] we describe a set of JViews-based plug-ins that provide a range of collaborative work facilities to JViews-based DSVL editing tools. A wide range of plug-ins are provided to support collaborative editing, awareness support, shared repositories, process-centred environment control and annotation of change histories, and so on. These

plug-ins use the aspect-based extensions to JViews and JComposer to support very dynamic, run-time plug-in support.

A generic approach to supporting diagram differencing and merging for collaborative design [57] presents a set of Pounamu and early Marama plug-ins that support collaborative work via DSVL diagram comparison and merging support. When working with other designers on a shared DSVL designs, such support is essential to enable changes made simultaneously or asynchronously to be compared, and selected changes to be “committed” to a shared, unified model. The techniques described in this paper are generic and work for any Pounamu or Marama DSVL-based tool.

DSVLs don’t have to be just two dimensional, box-and-line diagrams. **A 3D Business Metaphor for Program Visualization** [61] proposes a highly novel DSVL for visualising project management information using a 3D “city” metaphor of buildings, streets and various annotations. Like DCTracVis, we reverse engineer detailed project management information, abstract it, then use the city metaphor to visualise the project management data. Forward engineering from these project management visualisations using MDE was proposed to allow restructuring of projects based on modifications made to the city-based DSVLs.

Finally, bringing much of the prior works in this part together, we describe “sketching-based” interfaces to DSVL tools in *Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool* [30]. This describes a set of Marama plug-ins enabling sketching-based input of sketched designs, much as one would using a whiteboard or piece of paper. The sketched diagrams are automatically formalised into underlying Marama DSVL models. These sketch-supporting plug-ins also work with Marama versions of our collaborative work supporting plug-ins to support distributed, collaborative, sketched-based DSVL tools! The techniques described in this paper are generic and work for any Marama DSVL-based tool.

1.5.6 Part 6 – End user Applications of DSVLs and MDE

While many of the tools we have produced are for software engineers to help in their work, we have also produced many DSVL- and MDE-based tools for end users, usually focused on very specific domains of work. In this part I overview several papers that support a diverse range of end users and diverse range of end user application domains.

Domain-specific visual languages for specifying and generating data mapping systems [40] describes several tools aimed at supporting complex data integration, most aimed at supporting end users in different domains e.g. business analysts, construction engineers, and eHealth system integrators. Tools described include the Orion Message Mapper and Form-based Mapper described previously. Some of the data integration tools described were designed and built with MViews and JViews frameworks.

In *A domain-specific visual language for report writing* [18], we describe a tool developed with an industrial partner, PRISM, who build sophisticated software solutions for the commercial print industry. One of their software systems is a DSL-based report generation tool, to be used by print industry experts. This paper describes a DSVL-based report writing designer and generator. A DSVL is used to specify complex print industry report layout and content. MDE approaches are used to generate the textual DSL language, which is then compiled and run by PRISM’s existing software. The DSVL- and MDE-based approach makes authoring, modifying and understanding these complex professional print industry reports easier, faster and more maintainable. PRISM commercialised the prototype toolset. Our DSVL-based report designer was implemented with Microsoft Visual Studio’s DSVL-based designers and code generators.

HorusHPC, described previously, is aimed at scientists wanting to parallelise software for high performance computing domains. **Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-driven Engineering** [3] describes its DSVLs and web-based toolset. It uses MDE to generate lower-level models and GPU C code skeletons. Scientists then complete these code skeletons by hand. Partial HorusHPC models can also be reverse engineered from existing C code.

A visual language and environment for enterprise system modelling and automation [52] describes Enterprise Modelling Language (EML) and its support tool, described earlier in this chapter. EML and

EMLTool were designed for enterprise service modelling and generation of service orchestration scripts. EML is a DSL using novel tree- and overlay-based metaphors. EMLTool supports creating EML models, and uses these EML models to generate BPML₄WS service orchestrations using Eclipse model transformer and code generation plug-ins. A third party BPEL analyser checks for problems in the specifications and uses a DSL to highlight these to the user. EMLTool was realised using our Marama meta-toolset.

Described previously, Statistical Design Language (SDL) and its supporting tool, SDLTool, are aimed at professional and amateur statisticians, and provide multiple DSLs for high-level to low-level statistical survey designs. These are described in *A suite of visual languages for model-driven development of statistical surveys and services*, *Journal of Visual Languages and Computing* [49]. MDE techniques are used to generate R scripts and web service implementations of specified survey technique descriptions for reuse. SDLTool was implemented using the Pounamu meta-tools, along with a number of specially designed Pounamu plug-ins.

Finally in this part of the thesis I describe the use of CoNVERT for information integration and complex visualisation, in the context of household travel data aggregation, harmonisation, integration and visualisation. The paper *Engineering Complex Data Integration and Harmonization Systems* [8] presents an industry collaboration with the AURIN project and Data61 to source, analyse, aggregate, harmonise, transform, integrate, and visualise complex household travel survey data from several Australian states. End users of the DSL-based tool are domain experts in human geography-based survey data, government planning, and use of diverse government planning data.

1.5.7 Part 7 – Future directions

The single paper in this part of the thesis, *Towards Human-Centric Model-Driven Software Engineering* [33], outlines a new research programme for more human-centric, model-driven software engineering. This research includes integrating diverse human characteristics into requirements-level and design-level DSLs and using these human characteristics during model-driven engineering. The MDE generators either generate different apps and web site pages tailored to different end users, or generate configuration data that can be used by the apps/web sites at run-time to tailor them to diverse end user human characteristics. This work builds heavily on the contributions of many of the papers presented earlier in the thesis. It attempts to address some of the limitations of these works, specifically the lack of modelling diverse end user characteristics, such as age, gender, language, culture, personality, emotions, etc in DSLs, and the lack of using this information during MDE to produce software better suited to diverse end user needs.

1.6 Evidence of Impact

1.6.1 Citations

The research community has utilised the results from the papers in this thesis to inform many other research projects by many leading groups internationally. These are illustrated by both number of citations to many of the works, as well as citation of the works by many leading research teams. For example, several papers have over 200 citations (Google Scholar), or the paper in this thesis and its earlier conference version having together well over 200 citations. Examples include papers on Inconsistency management in MViews and Views [32], collaborative cloud security modelling and enactment [4], design pattern modelling and instantiation [56], collaborative DSL diagram diffing and merging [57], and aspect-oriented requirements engineering [25]. Many other papers, or the papers in this thesis and earlier conference version, have well over 100 citations. Examples include JViews/JComposer, Pounamu and Marama meta-tools papers and their earlier conference versions [35, 73, 37], MaramaAIC essential use case DSL-based tool [46], MaramaMTE web application testbed modeller and generator [19], Serendipity process-centred environment [39], Serendipity-II workflow modelling and enactment toolset [36], City metaphor project management information visualisation tool [61], and JViews-based collaborative editing plug-ins [28].

1.6.2 Industrial collaborations and Translation to practice

Many of the research works in this thesis have been carried out in collaboration with a wide range of industrial partners. In addition, a number of the meta-tools have been used on follow-on projects with these or other collaborators.

We have developed and used our data mapping tools, including the Orion Message Mapper, Form-based Mapper [40] and CoNVERT [7], with several companies interested in complex data integration problems. These include Orion Health, Peace Software, XSOL, First Data Utilities, NICTA, VicRoads and AURIN. The Orion Message mapper was a prototype for the very successful Rhapsody message mapping and integration toolset produced and commercialised by Orion Health.

We have used our process-modelling and workflow-modelling tools, including Serendipity and Serendipity-II [39, 36] derivatives of these, or our meta-tools Pounamu and Marama [73, 37], to implement similar prototype systems, for several domains. This includes modelling complex business process models with XSOL and Peace Software. The city visualisation DSVL and prototype tool were developed in collaboration with Peace Software's R&D team [61]. A summer student used Serendipity-II ideas to prototype a workflow tool with Peace Software to model and co-ordinate complex billing system processes. This was subsequently commercialised by Peace development teams.

We used our performance engineering tools, including Softarch/MTE and MaramaMTE [26, 19], on several industrial projects, several of them as confidential consulting projects on large scale enterprise system performance analysis and improvement. Some projects we are able to talk about include performance engineering of virtual database-based systems with XSOL, large scale clinical data repository engineering with Orion Health, and large scale database systems performance engineering with First Data Utilities. We collaborated with CSIRO on Softarch/MTE development [26]. This included using information from CSIRO collaborations with a range of large Australian corporates with complex enterprise system performance engineering needs. A patent for the principles underpinning Softarch/MTE was successfully applied for.

We collaborated with Swiss consulting company Sofismo AG to attempt a commercialisation of a derivative of the Marama meta-toolset [37]. This was to underpin Sofismo's complex system modelling and analysis work, carried out with a wide range of predominantly European corporates.

TeeVML development [54] was informed by our collaboration with CA Labs on generating complex enterprise system emulation environments. The idea was to augment CA Lab's commercial toolsets with support for very large scale testing environment modelling with TeeVML DSVL-based models, and then generate test bed emulation environments from these models using MDE techniques.

We have collaborated with several scientific and medical discovery teams using, among others, HorusHPC [3] and BiDaML [47]. These included the astrophysics team at Swinburne University of Technology on modelling complex radio telescope data process software for pulsar discovery, and the medical imaging team at the Alfred Hospital on MRI image processing for disease identification.

We collaborated with Thales on developing improved requirements engineering tools for complex air traffic control and related systems. This included modelling and semi-formalising textual requirements using DSVL models using MaramaAIC [46].

We have worked with several companies during the development and evaluation of BiDaML [47]. This included real estate cost estimation algorithms with ANZ bank, predicting congestion for Melbourne CBD using VicRoads data, and recently with eHealth application developers.

Our report writing DSVL-based tool was commercialised by PRISM [18]. This involved turning the initial prototype into an "industrialised" version, supporting larger scale reports, multiple user editing, and versioning.

For our Visual Wiki and VikiBuilder [44] work we successfully applied for a US patent for its underlying principles. We then secured very significant venture capital funding to commercialise this as the Mohio information visualisation platform.

1.6.3 Next Generation Education and Training

The impact I am most proud of from this body of work is the number and range of students, post-doctoral fellows and research assistants who I have worked with and educated in this domain. We used Pounamu and Marama tools for several years in our undergraduate final year software engineering course, many undergraduate Honors student projects, and in several graduate courses on domain-specific visual languages at the University of Auckland. While not having precise number of student teams or individual projects, well over 200 students used the tools to learn about DSVL and MDE principles and built their own DSVL- and MDE-tools.

Just contributing to the papers contained in this thesis alone, there are a total of 14 PhD students, 10 Masters by research students and 1 Honors student, and 10 post-doctoral fellows and 3 research assistants. Most of these students, research assistants and post-doctoral fellows have gone into industry positions with this knowledge and skills. Of the rest, 6 have academic positions and 4 have post-doctoral fellow positions as I write this. Including the many other derivative works using JViews/JComposer, Pounamu, Marama, and a few using third party frameworks and toolkits, there are a great many more students, research assistants and post-doctoral fellows who have benefited from learning in the environment and with the techniques and toolsets that we have created.

References

- [1] Agarwal, R. & Sinha, A. P. (2003). Object-oriented modeling with uml: a study of developers' perceptions. *Communications of the ACM*, 46(9), 248–256.
- [2] Ali, N. M. (2007). A generic visual critic authoring tool. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)* (pp. 260–261).: IEEE.
- [3] Almorsy, M. & Grundy, J. (2015). Supporting scientists in re-engineering sequential programs to parallel using model-driven engineering. In *Software Engineering for High Performance Computing in Science (SE4HPCS), 2015 IEEE/ACM 1st International Workshop on* (pp. 1–8).: IEEE.
- [4] Almorsy, M., Grundy, J., & Ibrahim, A. S. (2011). Collaboration-based cloud computing security management framework. In *2011 IEEE 4th International Conference on Cloud Computing* (pp. 364–371).: IEEE.
- [5] Almorsy, M., Grundy, J., & Ibrahim, A. S. (2014a). Adaptable, model-driven security engineering for saas cloud-based applications. *Automated software engineering*, 21(2), 187–224.
- [6] Almorsy, M., Grundy, J., & Rüegg, U. (2014b). Horuscml: Context-aware domain-specific visual languages designer. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 133–136).: IEEE.
- [7] Avazpour, I., Grundy, J., & Grunske, L. (2015). Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations. *Journal of Visual Languages & Computing*, 28, 195–211.
- [8] Avazpour, I., Grundy, J., & Zhu, L. (2019). Engineering complex data integration, harmonization and visualization systems. *Journal of Industrial Information Integration*, 16, 100103.
- [9] Bachman, C. W. (1969). Data structure diagrams. *ACM SIGMIS Database: The DATABASE for Advances in Information Systems*, 1(2), 4–10.
- [10] Barnett, S., Avazpour, I., Vasa, R., & Grundy, J. (2019). Supporting multi-view development for mobile applications. *Journal of Computer Languages*, 51, 88–96.
- [11] Barnett, S., Vasa, R., & Grundy, J. (2015). Bootstrapping mobile app development. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2 (pp. 657–660).: IEEE.
- [12] Blackwell, A. F. (2001). Pictorial representation and metaphor in visual language design. *Journal of Visual Languages & Computing*, 12(3), 223–252.
- [13] Böhm, C. & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5), 366–371.
- [14] Chaudron, M. R., Heijstek, W., & Nugroho, A. (2012). How effective is uml modeling? *Software & Systems Modeling*, 11(4), 571–580.
- [15] Chen, X., Hosking, J., Grundy, J., & Amor, R. (2018). Dctracvis: a system retrieving and visualizing traceability links between source code and documentation. *Automated Software Engineering*, 25(4), 703–741.

- [16] Cook, S., Jones, G., Kent, S., & Wills, A. C. (2007). *Domain-specific development with visual studio dsl tools*. Pearson Education.
- [17] Cox, P. T., Giles, F., & Pietrzykowski, T. (1989). Prograph: a step towards liberating programming from textual conditioning. In *Visual Languages, 1989., IEEE Workshop on* (pp. 150–156): IEEE.
- [18] Dantra, R., Grundy, J., & Hosking, J. (2009). A domain-specific visual language for report writing using microsoft dsl tools. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 15–22): IEEE.
- [19] Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., & Weber, G. (2006). Realistic load testing of web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06)* (pp. 11–pp): IEEE.
- [20] Esser, R. & Janneck, J. W. (2001). A framework for defining domain-specific visual languages. In *Workshop on Domain Specific Visual Languages, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2001)*.
- [21] File, P., Castell, A., Marshall, I., Walker, I., & Williams, L. (1970). From requirements specification to entity-relationship diagrams using rules. *WIT Transactions on Information and Communication Technologies*, 7.
- [22] Fowler, M. (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.
- [23] Green, T. R. G. & Petre, M. (1996). Usability analysis of visual programming environments: a ?cognitive dimensions? framework. *Journal of Visual Languages & Computing*, 7(2), 131–174.
- [24] Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education.
- [25] Grundy, J. (1999). Aspect-oriented requirements engineering for component-based software systems. In *Proceedings IEEE International Symposium on Requirements Engineering (Cat. No. PR00188)* (pp. 84–91): IEEE.
- [26] Grundy, J., Cai, Y., & Liu, A. (2005). Softarch/mte: Generating distributed system test-beds from high-level software architecture descriptions. *Automated Software Engineering*, 12(1), 5–39.
- [27] Grundy, J. & Hosking, J. (2002a). Developing adaptable user interfaces for component-based systems. *Interacting with computers*, 14(3), 175–194.
- [28] Grundy, J. & Hosking, J. (2002b). Engineering plug-in software components to support collaborative work. *Software: Practice and Experience*, 32(10), 983–1013.
- [29] Grundy, J. & Hosking, J. (2003). Softarch: Tool support for integrated software architecture development. *International Journal of Software Engineering and Knowledge Engineering*, 13(02), 125–151.
- [30] Grundy, J. & Hosking, J. (2007). Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool. In *29th International Conference on Software Engineering (ICSE'07)* (pp. 282–291): IEEE.
- [31] Grundy, J., Hosking, J., Cao, S., Zhao, D., Zhu, N., Tempero, E., & Stoeckle, H. (2007). Experiences developing architectures for realizing thin-client diagram editing tools. *Software: Practice and Experience*, 37(12), 1245–1283.

- [32] Grundy, J., Hosking, J., & Mugridge, W. B. (1998a). Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11), 960–981.
- [33] Grundy, J., Khalajzadeh, H., & McIntosh, J. (2020). Towards human-centric model-driven software engineering. In *ENASE* (pp. 229–238).
- [34] Grundy, J., Mugridge, R., Hosking, J., & Kendall, P. (2001). Generating edi message translations from visual specifications. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)* (pp. 35–42).: IEEE.
- [35] Grundy, J., Mugridge, W., & Hosking, J. (2000). Constructing component-based software engineering environments: issues and experiences. *Information and Software Technology*, 42(2), 103–114.
- [36] Grundy, J. C., Apperley, M. D., Hosking, J. G., & Mugridge, W. B. (1998b). A decentralized architecture for software process modeling and enactment. *IEEE Internet Computing*, 2(5), 53–62.
- [37] Grundy, J. C., Hosking, J., Li, K. N., Ali, N. M., Huh, J., & Li, R. L. (2012). Generating domain-specific visual language tools from abstract visual specifications. *IEEE Transactions on Software Engineering*, 39(4), 487–515.
- [38] Grundy, J. C. & Hosking, J. G. (1996). Constructing integrated software development environments with mvviews. *International Journal of Applied Software Technology*, 2(3-4), 133–160.
- [39] Grundy, J. C. & Hosking, J. G. (1998). Serendipity: integrated environment support for process modelling, enactment and work coordination. In *Process Technology* (pp. 27–60). Springer.
- [40] Grundy, J. C., Hosking, J. G., Amor, R., Mugridge, W. B., & Li, Y. (2004). Domain-specific visual languages for specifying and generating data mapping systems. *Journal of Visual Languages & Computing*, 15(3-4), 243–263.
- [41] Guerra, E., de Lara, J., Malizia, A., & Díaz, P. (2009). Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information and Software Technology*, 51(4), 769–784.
- [42] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3), 231–274.
- [43] Henderson, K. (1999). *On line and on paper: Visual representations, visual culture, and computer graphics in design engineering*. JSTOR.
- [44] Hirsch, C., Hosking, J., & Grundy, J. (2010). Wikibuilder: end-user specification and generation of visual wikis. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (pp. 13–22).
- [45] Huh, J., Grundy, J., Hosking, J., Liu, K., & Amor, R. (2009). Integrated data mapping for a software meta-tool. In *2009 Australian Software Engineering Conference* (pp. 111–120).: IEEE.
- [46] Kamalrudin, M., Hosking, J., & Grundy, J. (2017). Maramaic: tool support for consistency management and validation of requirements. *Automated software engineering*, 24(1), 1–45.
- [47] Khalajzadeh, H., Simmons, A. J., Abdelrazek, M., Grundy, J., Hosking, J., & He, Q. (2020). An end-to-end model-based approach to support big data analytics development. *Journal of Computer Languages*, 58, 100964.
- [48] Khambati, A., Grundy, J., Warren, J., & Hosking, J. (2008). Model-driven development of mobile personal health care applications. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (pp. 467–470).: IEEE.

- [49] Kim, C. H., Grundy, J., & Hosking, J. (2015). A suite of visual languages for model-driven development of statistical surveys and services. *Journal of Visual Languages & Computing*, 26, 99–125.
- [50] Kim, C. H., Hosking, J., & Grundy, J. (2005). A suite of visual languages for statistical survey specification. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)* (pp. 19–26).: IEEE.
- [51] Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing domain-specific design environments. *Computer*, 34(11), 44–51.
- [52] Li, L., Grundy, J., & Hosking, J. (2014). A visual language and environment for enterprise system modelling and automation. *Journal of Visual Languages & Computing*, 25(4), 253–277.
- [53] Li, Y., Grundy, J., Amor, R., & Hosking, J. (2002). A data mapping specification environment using a concrete business form-based metaphor. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (pp. 158–166).: IEEE.
- [54] Liu, J., Grundy, J., Avazpour, I., & Abdelrazek, M. (2016). A domain-specific visual modeling language for testing environment emulation. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 143–151).: IEEE.
- [55] Ludewig, J. (2003). Models in software engineering—an introduction. *Software and Systems Modeling*, 2(1), 5–14.
- [56] Maplesden, D., Hosking, J. G., & Grundy, J. C. (2001). A visual language for design pattern modelling and instantiation. In *HCC* (pp. 338–339).: Citeseer.
- [57] Mehra, A., Grundy, J., & Hosking, J. (2005). A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 204–213).
- [58] Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4), 316–344.
- [59] Moody, D. (2009). The “physics” of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.*, 35(6), 756–779.
- [60] Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1), 97–123.
- [61] Panas, T., Berrigan, R., & Grundy, J. (2003). A 3d metaphor for software production visualization. In *Proceedings on Seventh International Conference on Information Visualization, 2003. IV 2003.* (pp. 314–319).: IEEE.
- [62] Parnas, D. L. (1969). On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 24th national conference* (pp. 379–385).
- [63] Pelechano, V., Albert, M., Muñoz, J., & Cetina, C. (2006). Building tools for model driven development. comparing microsoft dsl tools and eclipse modeling plug-ins. In *DSDM*.
- [64] Petre, M. (2013). Uml in practice. In *2013 35th international conference on software engineering (icse)* (pp. 722–731).: IEEE.
- [65] Planas, E. & Cabot, J. (2020). How are uml class diagrams built in practice? a usability study of two uml tools: Magicdraw and papyrus. *Computer Standards & Interfaces*, 67, 103363.

- [66] Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE Computer Society*, 39(2), 25.
- [67] Shneiderman, B. (1993). 1.1 direct manipulation: a step beyond programming languages. *Sparks of innovation in human-computer interaction*, 17, 1993.
- [68] Shu, N. C. (1988). *Visual programming*. Van Nostrand Reinhold New York.
- [69] Sprinkle, J. & Karsai, G. (2004). A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3), 291–307.
- [70] Thiagarajan, R. K., Srivastava, A. K., Pujari, A. K., & Bulusu, V. K. (2002). Bpml: a process modeling language for dynamic business models. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2002.(WECWIS 2002). Proceedings. Fourth IEEE International Workshop on* (pp. 222–224).: IEEE.
- [71] Tolvanen, J.-P. & Rossi, M. (2003). Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 92–93).: ACM.
- [72] Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1), 109–142.
- [73] Zhu, N., Grundy, J., Hosking, J., Liu, N., Cao, S., & Mehra, A. (2007). Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, 80(8), 1390–1407.

2

DSVL Modelling Tool Development

2.1 Constructing component-based software engineering environments: issues and experiences

Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *Information and Software Technology*, Vol 42, No. 2, Elsevier, January 2000, Pages 103-114.

DOI: [10.1016/S0950-5849\(99\)00084-1](https://doi.org/10.1016/S0950-5849(99)00084-1)

Abstract: Developing software engineering tools is a difficult task, and the environments in which these tools are deployed continually evolve as software developers' processes, tools and tool sets evolve. To more effectively develop such evolvable environments, we have been using component-based approaches to build and integrate a range of software development tools, including CASE and workflow tools, file servers and versioning systems, and a variety of reusable software agents. We describe the rationale for a component-based approach to developing such tools, the architecture and support tools we have used some resultant tools and tool facilities we have developed, and summarise the possible future research directions in this area.

My contribution: Developed initial ideas for the approach, co-led design of the approach, implemented most of the software, led evaluation of the platform, co-authored substantial parts of the paper, investigator for funding for the work from the Foundation for Research Science and Technology (FRST)

Constructing Component-based Software Engineering Environments: Issues and Experiences

John Grundy¹, Warwick Mugridge², John Hosking²

¹ *Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand, Ph: +64-7-838-4452, Fax: +64-7-838-4155, jgrundy@cs.waikato.ac.nz*

² *Department of Computer Science, University of Auckland, Private Bag, Auckland, New Zealand, Ph: +64 9 3737599, Fax: +64 9 3737453, {john,rick}@cs.auckland.ac.nz*

Abstract

Developing software engineering tools is a difficult task, and the environments in which these tools are deployed continually evolve as software developers' processes, tools and tool sets evolve. To more effectively develop such evolvable environments, we have been using component-based approaches to build and integrate a range of software development tools, including CASE and workflow tools, file servers and versioning systems, and a variety of reusable software agents. We describe the rationale for a component-based approach to developing such tools, the architecture and support tools we have used, some resultant tools and tool facilities we have developed, and summarise possible future research directions in this area.

Keywords: component-based software architectures, multiple views, consistency management, tool integration, task automation

1. Introduction

Software engineering tools are usually complex applications. Many require multiple view support with appropriate consistency management techniques, most need to support multiple user facilities, and all need to support appropriate degrees of integration with other, often third party, tools. Software developers often want to reuse existing tools as well as enhance these tools or develop new tools as appropriate. Integrating a development team's tool set is often essential in order for the team members to use the different tools effectively on a project. It is a challenging task to provide these capabilities while ensuring that any resulting development environment is effective [3, 29, 26].

Many approaches to developing software tools exist, including the use of class frameworks [12], databases [8, 10], file-based integration [6], message-based integration [34], and canonical representations [25, 26]. We have been using a component-based approach to develop, enhance and integrate software tools [16, 17]. Our component-based software architecture includes abstractions useful for software tool development. Meta-CASE tools assist in designing and generating tools that use this architecture. We have developed a variety of useful software engineering tools using our tool set. Our experiences to date have shown that software tools developed with component-based architectures are

generally easier to: enhance and extend, integrate with other tools and deploy than tools developed using other approaches.

In the following section we review a variety of approaches to software tool construction, identifying their respective strengths and weaknesses. We then briefly characterise component-based software engineering tools, and provide an overview of our approach to developing such tools. Following this, we focus on how our approaches provide useful abstractions for building tools with support for multiple views, multiple users, tool integration, and task automation. We conclude by summarising our experiences in building and using component-based software engineering tools and outline possible future directions for research.

2. Related Work

Many approaches exist for building and integrating software engineering tools. In the following discussion we use Meyers' taxonomy to loosely group a variety of tools and their architectures, including file-based, message-passing (both local and distributed), database and canonical representation approaches [29]. We briefly identify the advantages and disadvantages of each for building tools and for supporting Tomas' taxonomy of types of integration (data, control, presentation and process) [39].

Many Unix-based software tools use a file system-based approach for integrating tools into an environment [6]. This allows tools to be very loosely integrated, so that building and adding new tools is straightforward. However, such tools generally provide limited data, control and presentation integration mechanisms.

The appearance of tight user interface and control integration of file-based tools can be achieved with message-passing approaches, as used by FIELD, HP Softbench and DEC FUSE [6, 21, 34]. These approaches allow existing tools to be wrapped and integrated into an environment very effectively. Data and process integration is generally not as well supported.

Many software tools are built using object-oriented frameworks, and use file or database-based persistency. These are often combined with version control and configuration management tools to support team development. Examples of such tools include VisualAge [23], EiffelCASE [24], PECAN [32], and Smalltalk-80 [12]. Such tools often provide very polished user interfaces and software development facilities, but are notoriously difficult to extend and integrate with third-party tools.

Tools using a database and database views to support data management and data viewing and editing include PCTE-based systems [4], SPADE [3] and EPOS [7]. The database provides a unifying data integration mechanism, but message-passing is often employed to facilitate control integration. Process-centred environments, such as SPADE, EPOS and ProcessWEAVER [11], support process integration by providing software process codification and execution facilities. The control of third-party tools is, however, difficult, as control integration with such tools is often very limited [9].

A variety of tool generation approaches have been developed. These typically produce database-integrated tools, such as KOGGE [10] and that of Backlund et al [2], or tools which use a canonical program representation, such as MultiView [1], Escalante [28] and Vampire [27]. Generated tools typically have good data, control and presentation integration, and can provide good process integration via the use of process-centered environments [3, 26]. However, integrating generated environments and tools produced using different architectures or generated by different systems has proved very difficult, resulting in limited presentation, data and control integration [3, 14, 26]. If collaborative work facilities are needed in such environments, they usually have to be built into the architecture and generator [3, 9, 13].

Some environments are designed to allow the addition of other tools. Examples include TeamWAVE [35], wOrlds [5], Oz [40], and Xanth [22]. The architectures of these environments usually focus on supporting control integration with other tools, and to a lesser extent presentation, data and process integration. Integration is typically limited to tools built with the same architecture, though Xanth and Oz provide quite flexible integration mechanisms for a wide range of tools.

3. Component-based SEEs

The component-based architectural approaches used by some software tools loosely correspond to a combination of message, database and canonical representation approaches. Components communicate via event propagation and/or message invocation, but often use other components to manage distributed data. Examples of environments using this approach include TeamWAVE, COAST [37], CoCoDoc [38], and SPE-Serendipity [14]. The use of software components to model tools and parts of tools typically allows more effective data and control integration than with other approaches. Presentation and process integration are also effectively supported if components are well designed for extension. Bridges between different component architectures and the use of Oz-style enveloping techniques allow component-based architectures to provide very effective third-party tool integration.

Component-based software engineering environments use a set of integrated components, with each component providing a tool or part of a tool used in the environment. Many of these tool components are reusable in other environments and possibly in other domains. Components are typically designed with a minimal knowledge of and dependency on other components, to facilitate reuse and deployment by plug-and-play. As a result, tools built using this model are typically highly reusable and externally controllable by other components. If carefully designed, these component-based tools thus tend to be more readily extended and integrated than tools developed using other approaches.

Figure 1 illustrates the concept of component-based software tools. This example environment consists of several components, which implement different software engineering tools and facilities. The editor, code generator and debugger share an abstract syntax tree (AST) and compiled code representation. The workflow tool co-ordinates use of these tools by monitoring events generated by them and by sending them messages. The workflow tool and data representation components use a distributed file server to manage shared data.

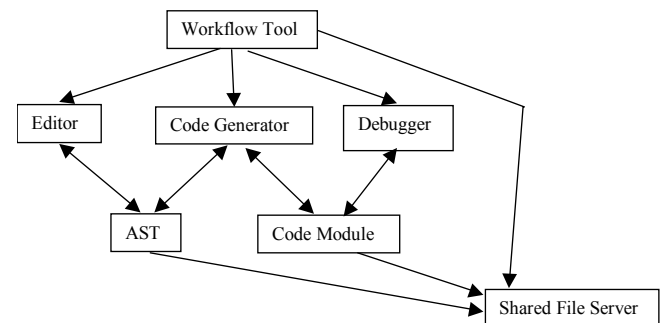


Figure 1. A simple component-based SEE.

Any of these tools or data representation components could be replaced another that satisfies the replaced interface (and required semantics).

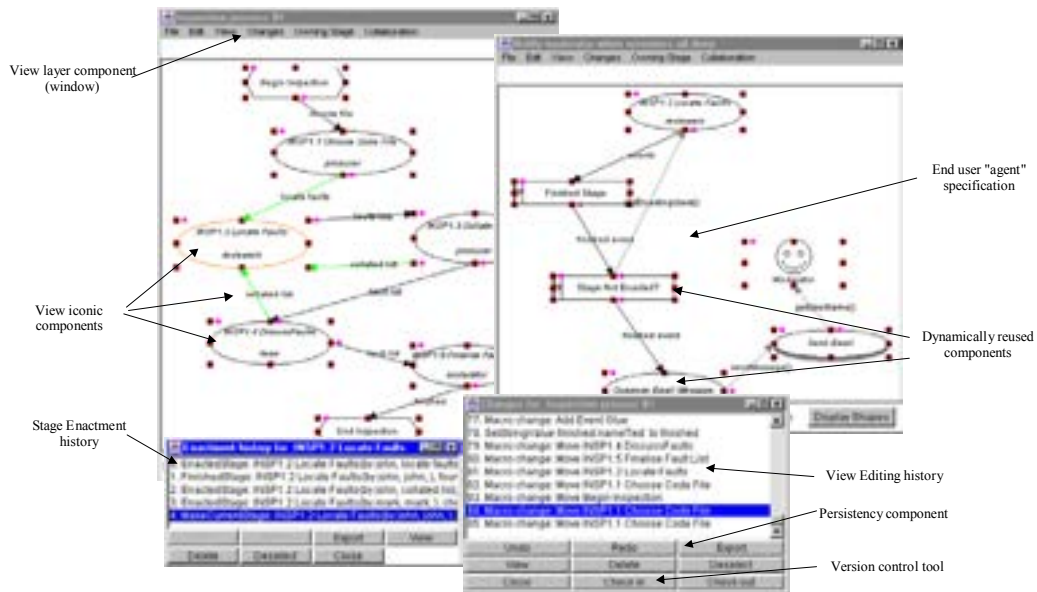


Figure 2. Serendipity-II: an example component-based software engineering tool.

New tools or components can be added, and may interact with existing components by monitoring events they generate or by sending them messages. Some of these tools could usefully be deployed in other domains. For example, the workflow tool and shared file server could be used in an Office Automation environment.

When designing and building component-based software engineering tools, a variety of issues must be addressed, including:

- Appropriate identification of components, allocation of data and behaviour to components and design of component interfaces.
- Design of common components for software tool abstractions such as repository management, multiple view, and collaborative work support.
- Appropriate design of components to permit external monitoring, control and user interface extension, thus supporting component reuse, enhancement and integration.
- Provision of appropriate architectures and tools so that tool developers can effectively design, build and deploy component-based tools.
- Provision of appropriate support to tool end users for deploying and integrating tools and enhancing environment behaviour via task automation agents.

Figure 2 shows the Serendipity-II software process modelling and enactment environment, a component-based software engineering tool that we have developed [18]. Serendipity-II illustrates how components can be designed, built and combined to build a sophisticated software engineering environment. Visual process modelling views provide editors and multiple views of process models, using components with extensible user interfaces and events that can be monitored. The graphical representation components are separated from the process model data representation components so that these can be developed independently. Task automation

agents are specified visually with components in the model reused by end users. Reusable components are used to implement repository management, multiple user editing and versioning support, event history management, and communication facilities.

In the following sections we outline our component-based approach to environment development, and demonstrate how that approach can be used to construct tools such as Serendipity-II, and describe our experiences in developing such tools.

4. Overview of Our Approach

JViews is a component-based software architecture and Java class framework which we have developed for implementing complex CASE tools, design environments and Information Systems [16]. JComposer is a component development environment which generates and reverse-engineers JViews components. This is used in conjunction with the BuildByWire iconic editor design tool [30], and the JVisualise run-time component visualisation and configuration tool [16], to design and implement JViews-based environments.

Jviews provides a set of abstractions for modelling and implementing software components. The JViews framework is implemented using JavaBeans [32]. JViews explicitly supports the design and implementation of applications with multiple, editable views of information, and multiple, distributed users. Many abstractions relating to these capabilities are lacking in other component-based toolkits.

Figure 3 shows the modelling of part of Serendipity-II in JViews. Components represent units of data and functionality that can be statically or dynamically linked to other components via relationships. A flexible event propagation and response model allows components to monitor other components and respond to state change or other events. Events from other components can be acted

on, but can also be modified, vetoed or prevented from reaching other components. Events can be stored to support modification histories, undo/redo and versioning. A variety of persistency, component distribution and collaborative editing and communication facilities are

provided by reusable JViews components. These make building new environments much easier, and also allow tools added to an existing environment to find and use components providing such facilities.

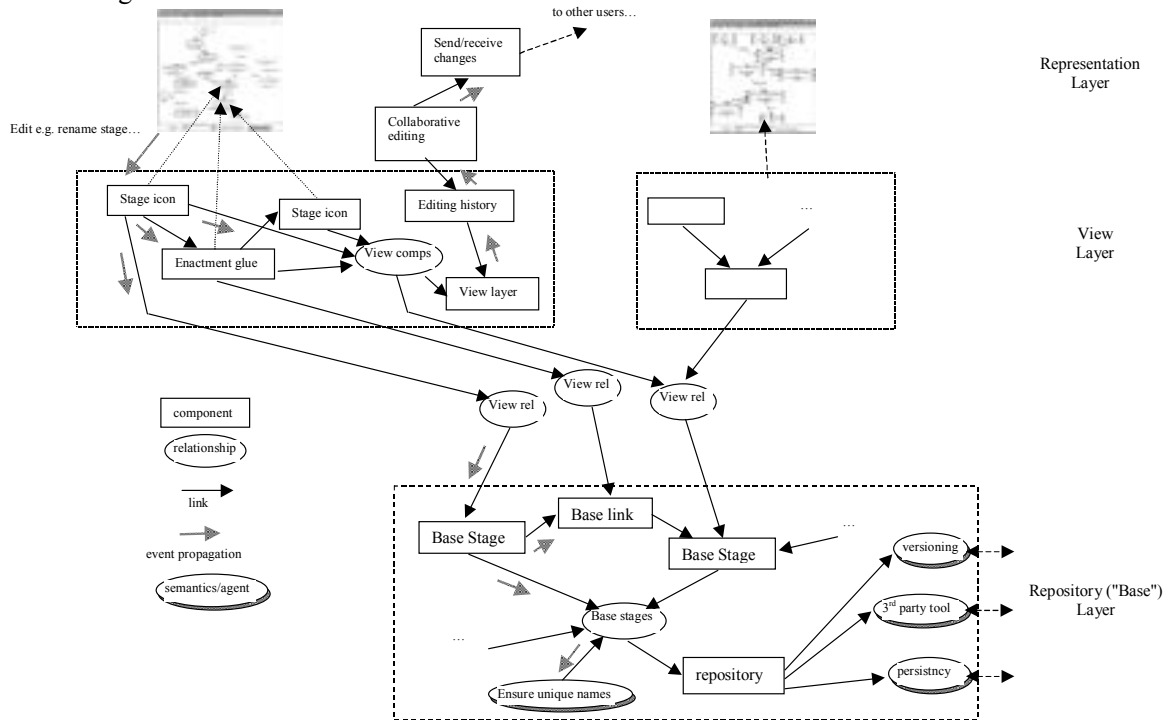


Figure 3. Some JViews components from Serendipity-II.

It is difficult to implement complex, component-based systems without good CASE tool support. We have developed JComposer to provide such support for systems developed with the JViews architecture. JComposer provides multiple views of JViews component specifications. It generates JViews classes to implement component models, and can be used to reverse engineer JViews classes into a high-level, visual architecture description language, similar to that used in Figure 3.

Windows (1) and (2) in Figure 4 show JComposer being used to model various Serendipity-II repository and view components. The component designer can specify information about components in various views, including functional and non-functional requirements, attributes, methods, relationships and events supported by the component, view mappings, and detailed code generation information. A novel event filter and action language allows static and dynamic event handling behaviour for components to be captured at a high level. This language was adapted in Serendipity-II to provide an agent specification and deployment language for end users. Various validation tests can be performed on a component model in JComposer to ensure generated JViews components are correct.

Windows (3) and (4) in Figure 4 show the BuildByWire iconic editor design and generation tool. BuildByWire allows software tool builders to design complex iconic representations and generate JavaBean

implementations of these and their editors. JComposer then allows tool developers to link JViews view components to these JavaBean components. The separation of presentation and view data has proved very effective in allowing tool developers to easily reuse iconic forms and replace view component iconic representations.

We have also developed a run-time component visualisation tool, JVisualise, that allows tool users to inspect and modify tool components using JViews component visual representations. Serendipity-II itself has been reused to provide a process modelling and enactment tool for JComposer and other JViews-based tools. Serendipity-II software agent specifications can include component representations from other tools that can be monitored or sent messages, facilitating control and process integration within an environment.

5. Multiple View Support

JViews represents a tool repository's data structure using components. Semantics are embodied in structural components or specified by additional components that monitor structural component events. Multiple views are supported by relationship components that link structural repository components to view components. Events generated due to modifications of the structural components are monitored by the relationship and forwarded to the view components and vice-versa.

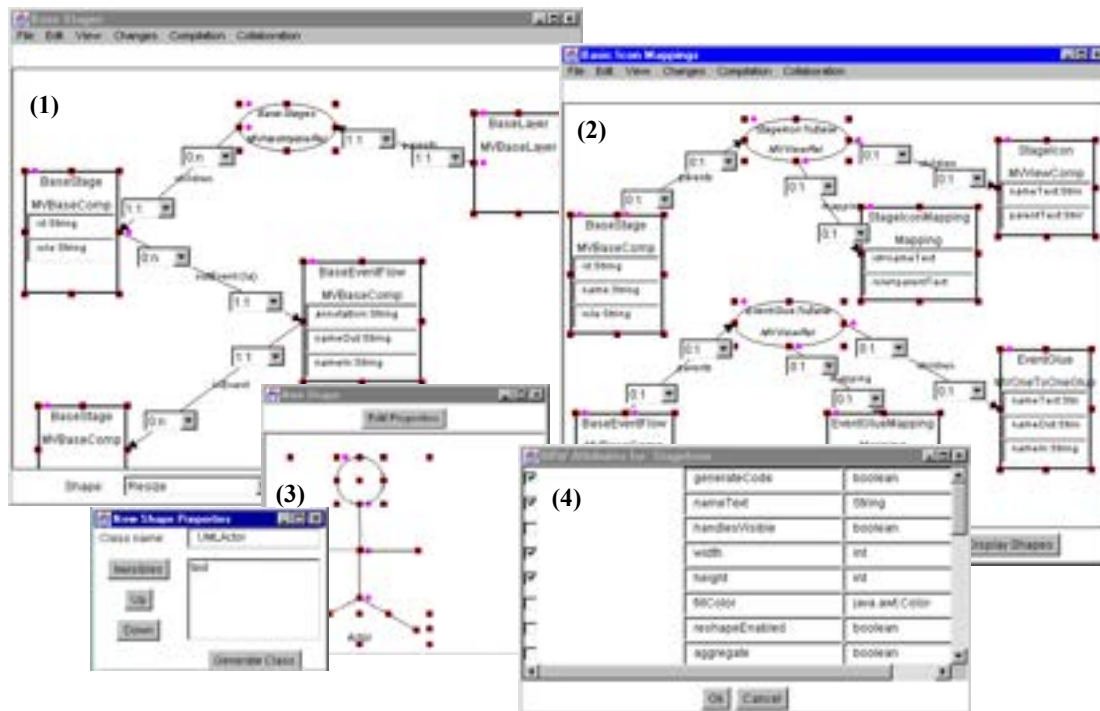


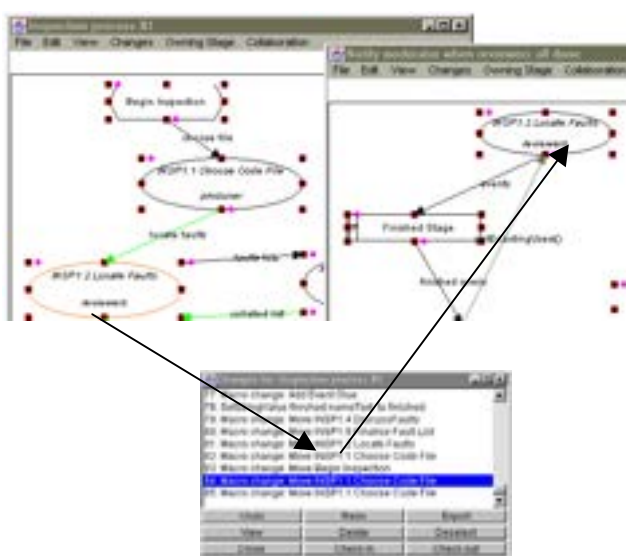
Figure 4. JComposer and BBW metaCASE tools in use.

Figure 5 shows how multiple views are modelled for parts of Serendipity-II using JViews. In Fig. 5(a) two process stage views contain reference to the same stage. An editing history for one view is also shown. Fig. 5(b) shows the JViews components corresponding to each view, and the shared repository, together with event flows when one of the process stage icons is modified. The event representing the change is recorded in the view's history component, and is also propagated via the view relationship to the repository component representing the process stage. This causes the repository component to update its state, thus generating events that are

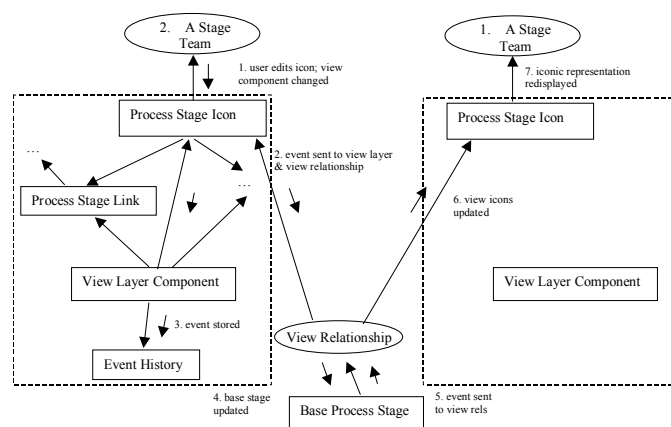
propagated to the corresponding process stage icon components in other views. The latter modify their state accordingly to maintain consistency with the initial view.

This is one example of a range of sophisticated inconsistency management facilities provided by JViews view components. These provide tool users with a range of techniques for keeping information consistent and for managing inconsistencies [9].

JViews view components also support extensible user interfaces; other components may modify their interfaces appropriately to support seamless user interface extension [9].



(a) Example of multiple views in Serendipity-II.



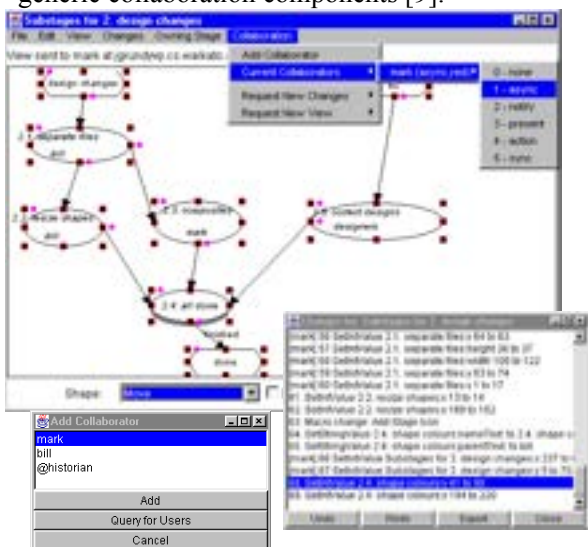
(b) JViews architecture supporting multiple views.

Figure 5. Multiple view support in JComposer.

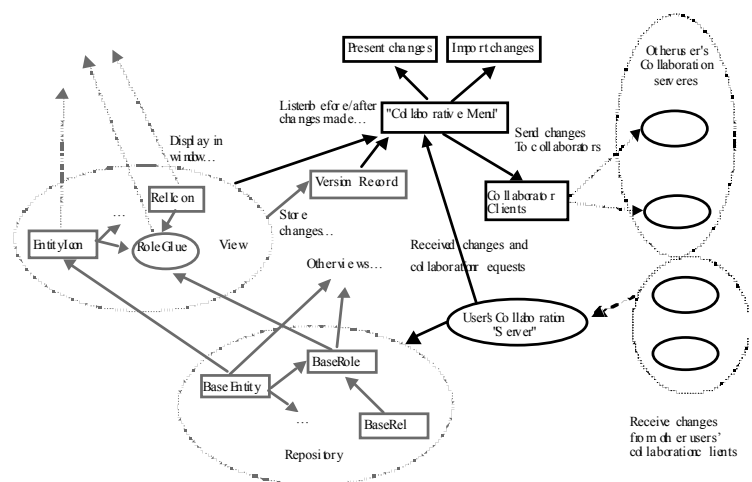
We have found the software component-based approach effective for supporting multiple views. New views and view components can be plugged into an environment without altering existing components. Our JViews components generate event objects that can be stored, undone and redone. Hence undo/redo and version control facilities can be achieved by adding components to manage these stored events. These event history components can be replaced, at run-time, with others, providing different facilities as necessary. Effective inconsistency management in large, multiple view software engineering environments is typically one of the most difficult features to provide. JViews' component based approach provides many useful abstractions that can be combined together simply to produce inconsistency management solutions that are much more difficult to provide with more conventional approaches [9].

6. Collaborative Work and Tool Integration Support

Nearly all multi-user software engineering tools build multiple user support into the tool as it is developed. In contrast, our component-based approach allows collaborative work-supporting facilities to be added as needed to a tool. This may even be done at run-time. JViews components broadcast state change events which can be monitored by other components, both before and after the stage change occurs (the former permitting veto of an operation before it takes effect). Synchronous and asynchronous editing facilities (including locking) can thus be added to existing JViews-based environments through the addition of extra collaboration components monitoring existing event sources and forwarding them to other environments. This can be done with no modification to the existing environment or to the added generic collaboration components [9].



(a) Example of asynchronous collaborative editing.



(b) Software components supporting collaborative editing.

Figure 6. Collaborative view editing and versioning support

Figure 6 (a) shows a "collaboration" menu in use in Serendipity-II to configure the "level" of collaborative editing with a colleague: asynchronous, synchronous and "presentation" (i.e. show editing changes to others as they occur but don't action them). The "change history" dialogue on the bottom, right hand side shows a history of editing events for the user's process model. Some changes were made by the user ("John"), and others by a collaborator ("Mark").

The illustration in Figure 6 (b) shows how these collaborative editing components were added to Serendipity-II. Such components may be added to any JViews-based environment, with no change to the components or the components that make up the environment.

A "collaboration menu" component is created when the user specifies that they want a view to be collaboratively edited. This component listens to editing changes in the view, and records them in a version record component. If the user is in presentation or synchronous editing mode with another user, the changes are propagated to that user's environment. This is achieved via a decentralised, point-to-point message exchanging system comprising of a "change sender" component and a "change receiver" component in each user's environment. A sender propagates view editing changes to each other users' environment who is interested in such a change i.e. all those who are collaboratively editing the view.

A user's environment receives view editing changes and passes them to the appropriate view collaborative editing component. This then stores and presents the received change in a dialogue (presentation mode editing) or actions it on the view (synchronous mode editing). In asynchronous editing mode, users request a list of changes made to another user's view and select, via a dialogue, those they wish to have applied to their own version of the view.

To support the replication of components (via object versioning), JViews has abstractions that are used to maintain copies of collaboratively edited views. When events that describe changes generated in one view are propagated to another user's environment any component references are translated appropriately.

Tool integration is supported in JViews-based environments in a similar manner to multiple views. Components which are part of one tool can request notification of events from components which are part of another tool, or can send these other tool components messages. This facilitates both control and data integration. JViews provides abstractions for identifying and communicating between components that are running in different virtual machine environments or that are resident on different physical machines.

Figure 7 shows a Serendipity-II component monitoring an event history component associated with a JComposer view. When the Serendipity-II component is

sent a modification event from the JComposer component, it passes this on to a component that determines if the event is of interest. If so, a third Serendipity-II component notifies the user of Serendipity-II.

Component-based software tools facilitate integration more readily than most other architectures for building tools, as components have well-defined interfaces with event monitoring mechanisms built in. Component-based tools also tend to be engineered for reuse and extension, allowing other components to externally control them and extend their user interfaces as necessary.

The success of this approach can be seen with Serendipity-II. It's component-based construction allows it to be seamlessly "bolted on" to any other JViews-based tool, providing that tool with integrated process modeling and enactment capabilities, without modification to the original tool [18].

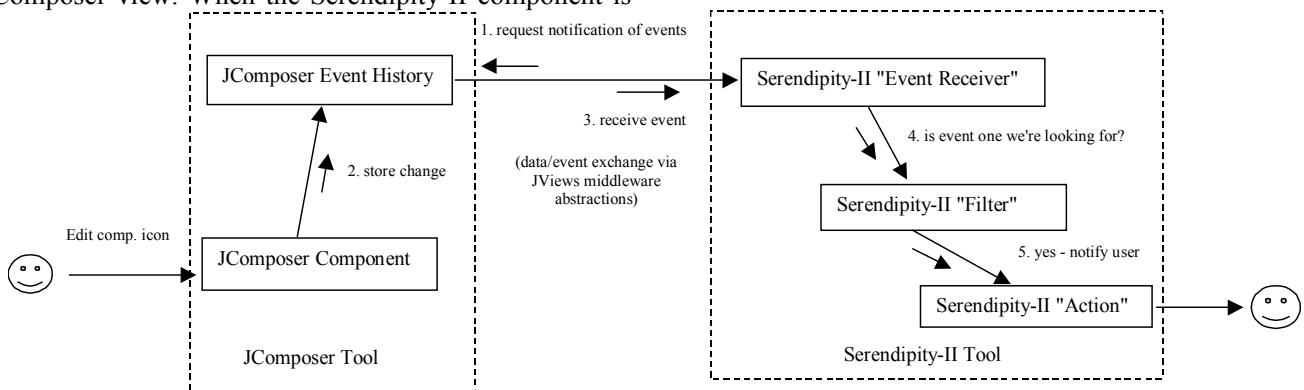


Figure 7. A simple tool integration example.

7. Environment Extension and Task Automation

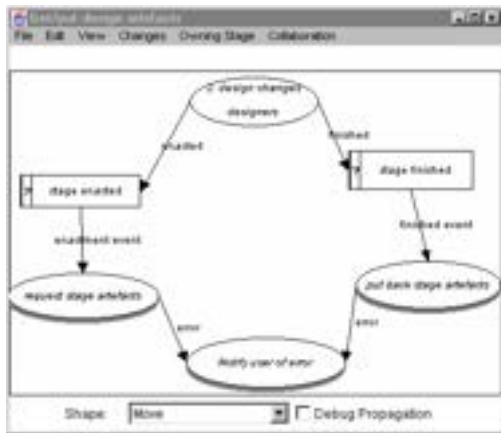
Software engineering environments need to support end user configuration of tools, extension of the environment, and automation of various tasks. For example, the user may wish to be notified of specific changes or to have additional constraints enforced. These capabilities permit the environment to be adapted to changing work processes and tool sets of software developers.

Serendipity-II allows for user enhancement via its visual event filtering and actioning language. This allows users to add, configure and link components into the environment. These event filtering and actioning models can then be deployed as "software agents" which modify an environment's composition and behaviour.

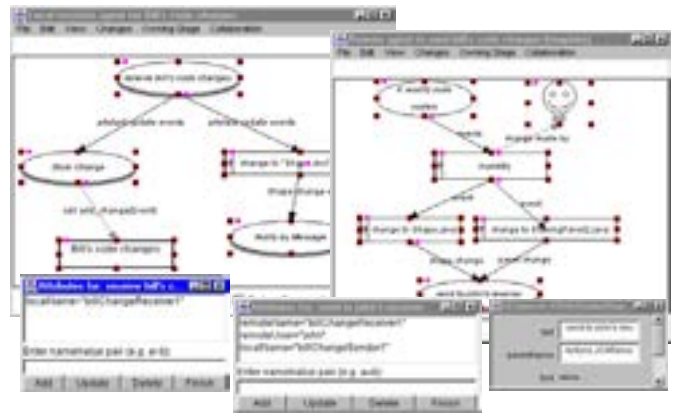
Figure 8 shows two examples of such agents: a simple task automation agent that also illustrates tool integration, and a distributed notification agent. The icons in these models represent Serendipity-II processes (ovals with

label and role name), event filters (square icons with question mark on left side), event actions (shaded icons with single label) and JViews components (square icons). Adding these icons to a Serendipity-II agent specification view creates appropriate JViews components that handle events or messages sent to the component. The example on the left instructs Serendipity to download or upload files to/from a shared file server when a particular process stage is enacted (started) or finished.

The example on the right has two parts. The agent on the left runs in user John's environment. The one on the right runs in Bill's environment, and gathers changes made by Bill to the "Shape" and DrawingPanel2" classes while doing the "modify code" process activity. These changes are forwarded by the "send to john's receiver" action to the "receive bill's code changes" action in the agent running in John's environment. John's agent stores the changes in a JViews history component "Bill's code changes" (left branch). It also notifies John by message if any change is made to the Shape class (right branch).



(a) Simple, local software agent for tool integration.



(b) Distributed software agent for notification.

Figure 8. Specifying simple task automation agents.

The component-based architecture of JViews-based environments allows such task automation and tool integration facilities to be straightforwardly built using our event filtering and actioning model. JViews components generate events which can be filtered or used to produce a wide variety of "actions" (notify user, abort operation, communicate with another tool/component, open/close views, store change event, etc.). We have built a variety of reusable JViews components to help facilitate the construction of software agents like those in Figure 7. These include components to: perform parameterised filtering of events; inter-machine event passing and operation invocation; store events; notify users of events and support user communication; and provide interfaces to various third party tools (e.g. MS Word™ and Excel™, Eudora™, Netscape™ and WinEdit™).

8. Future Research Opportunities

Our experiences with component-based software engineering tools have so far been very positive. Environments like JComposer and Serendipity-II were easier to build than comparable earlier systems we developed without using components [15]. This has been due to the high degree of reusability of our JViews components, the useful set of middleware and user interface abstractions embodied by JViews components, and the use of the JComposer and BuildByWire meta-CASE tools. We have been able to extend environments like Serendipity-II using our visual event filtering and actioning facility. This has allowed us to easily develop and deploy new reusable components for tool integration and environment extension, without having to substantially modify existing structures and behaviour.

As more software tools begin to utilise component-based architectures, like JavaBeans [32] and COM [36], and distributed object management facilities like CORBA [31], it becomes easier to effectively integrate such tools with JViews-based environments. Better middleware components to support collaborative work, data

persistence, distributed and parallel processing and remote notification will allow better performing software tools to be built. If appropriate component interface designs have been used for such middleware capabilities, upgrading environment infrastructures also becomes easier and more successful.

A problem we encountered with some JViews and BuildByWire components was poor extensibility of their user interfaces. Components need to be designed so that their user interfaces can be appropriately extended by other components, to ensure a consistent look-and-feel and to tailor the user interface to suits the needs of subsets of users.

This becomes more difficult as highly reusable components are developed whose application domains are not fully known during their design, and when third-party component-based tools are reused. Similar issues arise with middleware component interface and capability design, requiring the development of better component-based system interface standards and design techniques.

Multiple view and tool integration support necessarily involves mapping operations and/or events from components in one view/tool into another. Support for complex inter-component event and operation mapping is lacking in most component-based systems, making development of multiple views and tools more difficult. This is an area we have attempted to address in our work, but additional work is needed.

Software developers are tending to require more control over the configuration of their tools, composition of their environments and behaviour of their tools. Appropriate end-user configuration facilities for component-based systems are thus essential to ensure they are effective. Similarly, the use of software agents to automate tasks and the effective cataloguing and retrieval of reusable components remain issues requiring further research.

9. Conclusions

Our experience in the development of complex component-based software engineering tools has convinced us of the value of a component-based approach. The modularity provided by component interfaces and the flexibility provided by the “plug and play” event-based composition of components have proven to be of considerable value in the both the development of such environments, and the provision of end users with the capability to tailor and extend the environments. Several issues remain to be solved to enable software components in general to be used on a large scale, and these also impact on our work on the generation of software engineering tools.

References

1. Altmann, R.A. and Hawke, A.N. and Marlin, C.D., An Integrated Programming Environment Based on Multiple Concurrent Views, *Australian Computer Journal* 20 (2), May 1988, 65-72.
2. Backlund, B. and Hagsand, O. and Pherson, B., Generation of Visual Language-oriented Design Environments, *Journal of Visual Languages and Computing* 1 (4), 1990, 333-354.
3. Bandinelli, S. and DiNitto, E. and Fuggetta, A., Supporting cooperation in the SPADE-1 environment, *IEEE Transactions on Software Engineering* 22 (12), December 1996, 841-865.
4. Bird, B., An Open Systems SEE Query Language, *Proceedings of 7th Conference on Software Engineering Environments*, Noordwijkerhout, Netherlands, April 5-7 1995, IEEE CS Press.
5. Bogia, D.P. and Kaplan, S.M., Flexibility and Control for Dynamic Workflows in the wOrlds Environment, *Proceedings of the Conference on Organisational Computing Systems*, Milpitas, CA, November 1995, ACM Press.
6. Champine, M.A. A visual user interface for the HP-UX and Domain operating systems, *Hewlett-Packard Journal* 42 (1), 1991, 88-99.
7. Conradi, R. Hagaseth, M., Larsen, J., Nguyen, M.N., Munch, B.P., Westby, P.H., Zhu, W. and Jaccheri, M.L., EPOS: Object Oriented Cooperative Process Modeling, In *Software Process Modeling & Technology*, A.Finkelstein and J. Kramer and B. Nuseibeh Eds, Research Studies Press, 1994.
8. Daberitz, D. and Kelter, U. Rapid Prototyping of Graphical Editors in an Open SDE, *Proceedings of 7th Conference on Software Engineering Environments*, Noordwijkerhout, Netherlands, April 5-7 1995, IEEE CS Press, pp. 61-73.
9. Di Nitto, E. and Fuggetta, A. Integrating process technology and CSCW, *Proceedings of IV European Workshop on Software Process Technology*, Leiden, Netherlands, April 1995, LNCS, Springer-Verlage.
10. Ebert, J. and Sutenbach, R. and Uhe, I. Meta-CASE in practice: A Case for KOGGE, *Proceedings of CaiSE*97*, Barcelona, Spain, June 10-12 1997, LNCS 1250, Springer-Verlage, pp. 203-216.
11. Fernström, C. ProcessWEAVER: Adding process support to UNIX, *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993, IEEE CS Press, pp. 12-26.
12. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Environment*, Addison-Wesley, Reading MA, 1984.
13. Grundy, J.C. Human Interaction Issues for User-configurable Collaborative Editing Systems, *Proceedings of APCHI'98*, Tokyo, Japan, July 15-17 1998, IEEE CS Press, pp. 145-150.
14. Grundy, J.C. and Hosking, J.G., Serendipity: integrated environment support for process modelling, enactment and work coordination, *Automated Software Engineering* 5 (1), January 1998.
15. Grundy, J.C. and Hosking, J.G. and Fenwick, S. and Mugridge, W.B. Connecting the pieces, Chapter 11 in *Visual Object-Oriented Programming*, Burnett, M., Goldberg, A., Lewis, T. Eds, Manning/Prentice-Hall, 1995.
16. Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Static and Dynamic Visualisation of Software Architectures for Component-based Systems, *Proceedings of SEKE'98*, San Francisco, June 18-20 1998, KSI Press.
17. Grundy, J.C., Hosking, J.G. and Mugridge, W.B., Coordinating distributed software development projects with integrated process modelling and enactment environments, *Proceedings of 7th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Palo Alto, June 17-19 1998, IEEE CS Press.
18. Grundy, J.C., Apperley, M.D., Mugridge, W.B. and Hosking, J.G. An architecture for decentralized process modelling, *IEEE Internet Computing*, September/October 1998.
19. Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Inconsistency management for multiple-view software development environments, *IEEE Transactions on Software Engineering* 24 (11), November 1998, 960-681.
20. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. Tool integration, collaboration and user interaction issues in component-based software architectures, *Proceedings of TOOLS Pacific'98*, Melbourne, Australia, Nov 24-26 1998, IEEE CS Press, pp. 289-302.
21. Hart, R.O. and Lupton, G., DECFUSE: Building a graphical software development environment from Unix tools, *Digital Tech Journal* 7 (2), 1995, 5-19.
22. Kaiser, G.E. and Dossick, S. Workgroup middleware for distributed projects, *IEEE WETICE'98*, Stanford, June 17-19 1998, IEEE CS Press, pp. 63-68.
23. IBM Corporation, IBM Visual Age for Java™, <http://www.software.ibm.com/ad/vajava>, 1997.
24. Interactive Software Engineering Inc., Eiffel CASE™, <http://www.eiffel.com/products/case.html>, 1998.
25. Magnusson, B. and Asklund, U. and Minör, S. Fine-grained Revision Control for Collaborative Software Development, *Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, Los Angeles, 1993, pp. 7-10.
26. Marlin, C. and Peuschel, B. and McCarthy, M. and Harvey, J., MultiView-Merlin: An Experiment in Tool Integration, *Proceedings of the 6th Conference on Software Engineering Environments*, IEEE CS Press, 1993.
27. McIntyre, D.W., Design and implementation with Vampire, In *Visual Object-Oriented Programming*, M. Burnett and A. Golberg and T. Lewis Eds, Manning Publications, Greenwich, CT, USA, 1995.

28. McWhirter, J.D. and Nutt, G.J., Escalante: An Environment for the Rapid Construction of Visual Language Applications, Proceedings of the 1994 IEEE Symposium on Visual Languages, IEEE CS Press, 1994.
29. Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software 8 (1), January 1991, 49-57.
30. Mugridge, W.B., Hosking, J.G. and Grundy, J.C. Vixels, CreateThroughs, DragThroughs and AttachmentRegions in BuildByWire, Proceedings of OZCHI98, Adelaide, Australia, November 30-Dec 4 1998, IEEE CS Press, pp. 320-327.
31. Object Management Group, OMG CORBA, <http://www.omg.org/>, 1998.
32. O'Neil, J. and Schildt, H. Java Beans Programming from the Ground Up, Osborne McGraw-Hill, 1998.
33. Reiss, S.P., PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering 11 (3), 1985, 276-285.
34. Reiss, S.P., Connecting Tools Using Message Passing in the Field Environment, IEEE Software 7 (7), July 1990, 57-66.
35. Roseman, M. and Greenberg, S., Simplifying Component Development in an Integrated Groupware Environment, Proceedings of the ACM UIST'97 Conference, ACM Press, 1997.
36. Sessions, R. COM and DCOM: Microsoft's vision for distributed objects, John Wiley & Sons, 1998.
37. Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. Designing object-oriented synchronous groupware with COAST, Proceedings of the ACM Conference on Computer Supported Cooperative Work, ACM Press, November 1996, pp. 21-29.
38. Ter Hofte, G.H. and van der Lugt, H.J. CoCoDoC: a framework for collaborative compound document editing based on OpenDoc and CORBA, Proceedings of the IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms, 26-30 May 1997, Toronto, Canada, pp. 15-33.
39. Thomas, I. and Nejme, B. Definitions of tool integration for environments, IEEE Software 9 (3), March 1992, 29-35.
40. Valetto, G. and Kaiser, G.E., Enveloping Sophisticated Tools into Process-centred Environments, Automated Software Engineering 3, 1996, 309-345.

2.2 Inconsistency Management for Multi-view Software Development Environments

Grundy, J.C., Hosking, J.G., Mugridge, W.B. Inconsistency Management for Multi-view Software Development Environments, *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, 1998, pp. 960 - 981.

DOI: [10.1109/32.730545](https://doi.org/10.1109/32.730545)

Abstract: Developers need tool support to help manage the wide range of inconsistencies that occur during software development. Such tools need to provide developers with ways to define, detect, record, present, interact with, monitor and resolve complex inconsistencies between different views of software artifacts, different developers and different phases of software development. This paper describes our experience with building complex multiple-view software development tools that support diverse inconsistency management facilities. We describe software architectures we have developed, user interface techniques used in our multiple-view development tools, and discuss the effectiveness of our approaches compared to other architectural and HCI techniques.

My contribution: Led development of the key ideas, co-designed the approach, implemented most of the software, led evaluation of the platform, wrote most of the paper, one of the investigators on grant for funding for the work from the Foundation for Research Science and Technology (FRST)

Inconsistency Management for Multiple-View Software Development Environments

John Grundy[†], John Hosking^{††} and Rick Mugridge^{††}

[†]Department of Computer Science, University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

^{††}Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john, rick}@cs.auckland.ac.nz

Abstract

Developers need tool support to help manage the wide range of inconsistencies that occur during software development. Such tools need to provide developers with ways to define, detect, record, present, interact with, monitor and resolve complex inconsistencies between different views of software artefacts, different developers and different phases of software development. This paper describes our experience with building complex multiple-view software development tools that support diverse inconsistency management facilities. We describe software architectures we have developed, user interface techniques used in our multiple-view development tools, and discuss the effectiveness of our approaches compared to other architectural and HCI techniques.

Keywords: inconsistency management, multiple views, integrated software development environments, collaborative software development

1. Introduction

Software developers work with a variety of specifications of software systems at differing levels of abstraction, including software requirements, analysis, design, implementation and documentation. Inconsistencies between parts of a specification, or between specifications at differing levels of abstraction, can arise during or between phases of software development. Over time, such inconsistencies must be resolved in order to produce a working software system, or partially resolved to produce part of a system for testing and quality assurance purposes. The use of disparate software development tools on a project by multiple developers is usually essential when developing today's complex software systems, but the use of such tools can exacerbate the creation of inconsistencies [44, 53].

Systems have their specifications split among several different tools, often used by different developers, with partial redundancy resulting. Modifying part of a system specification in one tool can thus introduce inconsistencies with related parts of the system specified in other tools, between specifications shared by different developers, or even cause inconsistencies to occur within the same tool. For example, one developer modifying a class interface design in an OOA/D tool will often cause other design diagrams in the same tool to become inconsistent, and cause parts of the system's design held in other tools to become inconsistent. It also may cause code based on these designs to become inconsistent, and the designs to become inconsistent with documentation and analysis diagrams developed by the same or other developers. In a complex system involving multiple developers and development tools, developers are very often unaware of the introduction, or even existence, of such inconsistencies.

The management of inconsistencies during a software development project can be improved when the tools used are tightly, or even loosely, integrated [53, 63, 27, 59]. In such systems, developers interact with multiple "views" (or perspectives) of software at the same and different levels of abstraction. Developers will usually have views partitioned into system requirements, analysis, design, implementation, documentation etc, and further views partitioning specifications at each level of abstraction [29, 63, 53].

Inconsistency detection is needed after one view is edited in order to detect: structural inconsistencies between views (e.g. class attributes added in one view aren't in another); semantic inconsistencies in specifications (e.g. a type mis-match between method calls or a non-existent method is called); inconsistencies between specifications at different levels of abstraction (e.g. system requirements and design conflict); and inconsistencies between the work of multiple developers (e.g. when any inconsistency is caused by different developers working concurrently).

Some inconsistencies may be automatically corrected, for example by tools updating the information in one view when another, related view has been edited. However, many inconsistencies can not, or should not, be automatically corrected. Hence mechanisms are required for tools to inform developers of inconsistencies, and developers require facilities to monitor and resolve inconsistencies. As it is usually impossible to keep a software system consistent at all times, multiple view tools should not be overly prescriptive in attempting to enforce consistency, except when instructed to do so by developers. Tools thus need to support the long-term management of inconsistencies. Having multiple developers adds to the complications of detecting, presenting and tracking inconsistencies made by others, and in negotiating resolutions to inconsistencies.

In this paper we focus on managing inconsistencies primarily between the analysis, design and implementation specifications of software in multiple-view and multiple-user integrated tools. Our main interest and contributions are in detecting structural, semantic, inter-person and inter-process inconsistencies as they occur, and allowing developers to manage these. This contrasts with the kind of "batch" inconsistency checking done by compilers. The contributions of our work include:

- an architecture for software development tools which supports the representation of software specifications, and the definition, detection, representation and propagation of inconsistencies between these specifications
- the realisation of this architecture in frameworks and tool generators for constructing multiple-view and multiple-person software development tools
- techniques for presenting inconsistencies to developers
- techniques that allow developers to monitor, negotiate and resolve inconsistencies
- tool configuration support allowing software developers to configure their environment's inconsistency management policies
- a range of exemplar software development tools built using our architectures that have been deployed on a variety of small- and medium-sized projects and which demonstrate the utility of our techniques.

We begin with various inconsistency problems which can arise in multiple view, multiple user environments. These problems are illustrated with an integrated software development environment that we have developed. We then discuss a range of existing techniques and systems that attempt to address some of these inconsistency management issues in software development tools. Section 4 describes our inconsistency management model, the software architecture that realises this model, and meta-tools we have developed to aid in the construction of software development tools. Section 5 provides a user's perspective on our approaches to the presentation of inconsistency, while Section 6 describes how a user can interact with such presentations to monitor and/or resolve them. Section 7 discusses inconsistency management during collaborative software development. Section 8 argues that inconsistency management configuration facilities are necessary and shows how they can be supported. Section 9 describes our experiences with our inconsistency management techniques and tools, and evaluates them. We conclude with the contributions of this research and directions for future work.

2. Problem Domain: Inconsistencies in Multiple-View Environments

2.1. An Example Multiple-View Software Development Environment

To illustrate the range of inconsistency management requirements of a multiple view software development environment, we introduce SPE (Smart Programming Environment). SPE is an integrated software development environment for developing object-oriented programs [27]. It supports multiple textual and graphical views of information, with full bi-directional consistency management between all views. For example, the same artefact can be viewed in several analysis, design, code and documentation views, as illustrated by the "customer class" specified in Figure 1 in views "root class" (OO design diagram) and view "customer - class Interface" (textual class interface code). We have integrated SPE and the Serendipity software process modelling and enactment environment [34] to support coordinated multiple-user software development in SPE. Figure 1 shows a simple software process for modifying a system design enacted in Serendipity ("aff2. Design, Code & Test-subprocess").

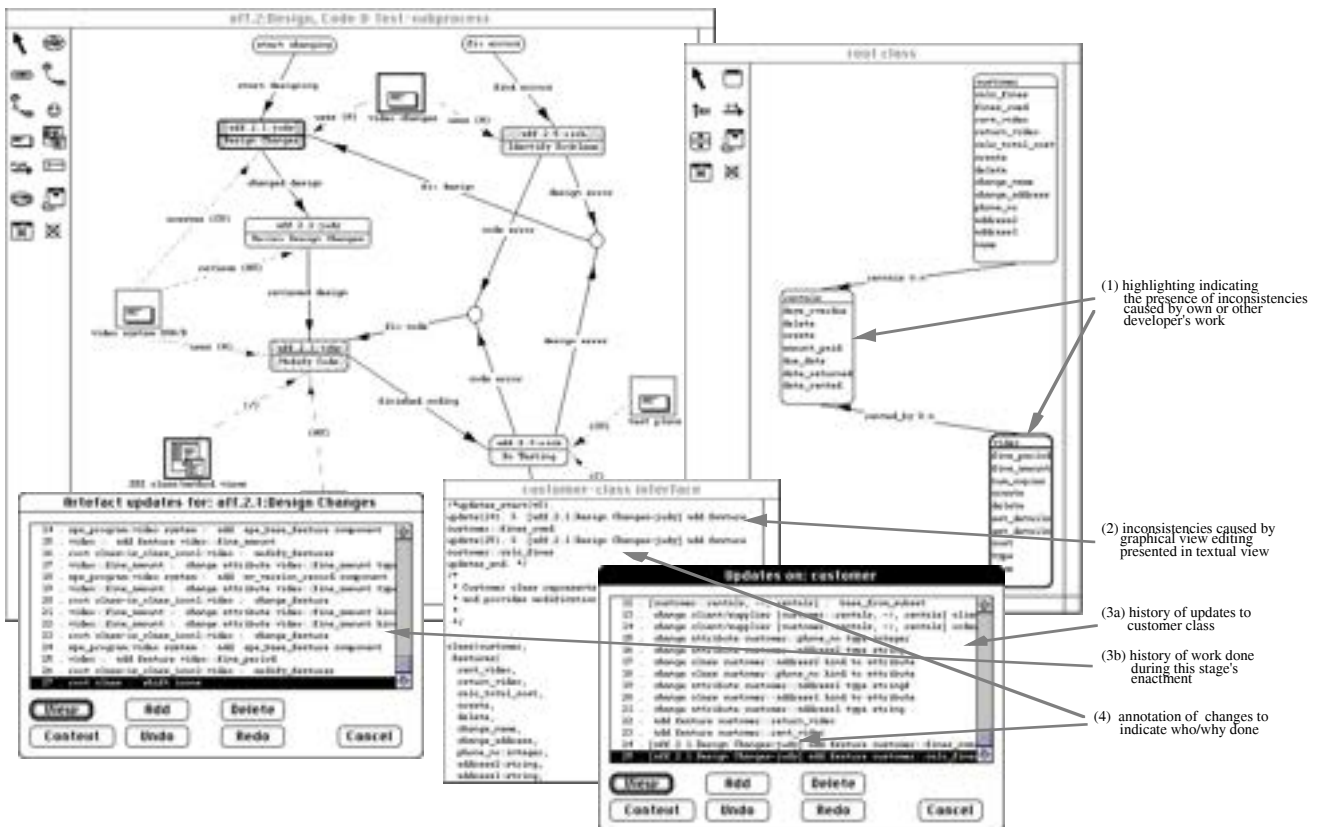


Figure 1. An example multi-view editing environment.

As information is shown in multiple views in SPE, and at differing levels of abstraction, a range of inconsistencies can occur when modifying different views of software specifications. For example, renaming a class in an OO design view means all other analysis, design, implementation and documentation views become inconsistent unless the class name is changed in these views. Such a change could be carried out automatically by SPE. However, adding a method call connection in an OO design view can not be automatically added to the appropriate textual code view(s) affected by this change, so the code views become inconsistent. SPE has no way of determining appropriate arguments to pass to the method, nor where in the textual code the method call should go. Similarly, changing the type of an attribute in a design view may allow some degree of automatic resolution, but often requires developers to change other design and/or implementation decisions.

When multiple developers share SPE views, they sometimes want to collaborate closely to analyse and negotiate about inconsistencies and how these should be resolved. Thus collaborative editing techniques are required to facilitate close collaboration and inconsistency management. However, at other times developers will work independently, modifying alternative versions of views. Any inconsistencies generated require tracking, and then negotiation and resolution, usually during version merging.

Figure 1 shows a few examples of how inconsistencies are managed in SPE:

- (1) Icons in views can be highlighted to indicate the presence of inconsistencies.
- (2) Inconsistency “descriptions” can be presented to developers in dialogues or within views to inform them of inconsistencies. Such descriptions may simply describe changes made to other views which affect the specification shown in the view, or may describe semantic constraint violations that need to be resolved. Presentations may also allow developers to “interact” with the inconsistency, for example to select an inconsistency and request more information about it or request a view be automatically modified to resolve the inconsistency.
- (3a) “Histories” of changes made to software specifications and
- (3b) during process stage enactment are kept, which also record and present semantic inconsistencies (e.g. type mis-matches, undefined variables and methods), and allow tracking of and interaction with inconsistencies.
- (4) Inconsistency descriptions presented to developers can be annotated with extra information e.g. relative importance, additional reasons why it has been detected, and Serendipity process model information.

2.2. Inconsistency Management Requirements

From our experience with developing SPE and many other multiple-tool and multiple-user environments [71,29] we have developed several key requirements for supporting inconsistency management in such tools:

- *Description of syntax and semantics.* The software architecture used to build such an environment must support the representation of a wide range of software specification and view structures and the semantics associated with these structures.
- *Inconsistency detection.* Inconsistencies must be detected when: i) software specifications are modified and related specifications can not be automatically updated to be kept consistent; ii) semantic constraints associated with modified software structures are violated in some way; or iii) the work of multiple developers interferes, causing structural or semantic inconsistencies.
- *Inconsistency representation.* As inconsistencies may range from short-lived to very long-term, ways are needed to represent such inconsistencies, propagate them among related software specifications, and record them for monitoring and later resolution.
- *Inconsistency reason information.* Inconsistencies occur for a reason. Associating the reason for a change can help a user in dealing with that inconsistency, especially when it was made by someone else and/or at a different time. This might simply be that a change has been made to an overlapping software specification [22], and hence the change needs to be incorporated in some way in all other affected specifications, or a semantic constraint has been violated, or another developer has made a change. The representation of inconsistencies used by an environment needs to support a description of: what was changed or what constraint was violated; who caused inconsistencies; the process stage or “context” inconsistencies occur in; reasons inconsistencies are detected; and the relative “importance” of inconsistencies, including during what phases of development an inconsistency can be “tolerated” before being resolved.
- *Inconsistency presentation.* Developers need to be informed of the presence of inconsistencies when using views and need to be provided with a variety of information about inconsistencies.
- *Inconsistency monitoring.* Developers need to monitor inconsistencies at different times and in different ways, and thus require facilities to query for specific kinds of inconsistencies and to have these presented appropriately.
- *Inconsistency interaction and resolution.* Developers need to interact with inconsistency presentations to locate their causes, gain more information about them and resolve them.
- *Inconsistency resolution negotiation.* Multiple developers require support for negotiating the resolution of inconsistencies affecting more than one person.
- *Inconsistency management configuration.* Developers require facilities to configure when and how inconsistencies are detected, monitored, stored, presented, and possibly automatically resolved.

3. Related Research

Many software architectures have been developed to aid in building tools that provide multiple views of software development. However, none that we are aware of completely support the range of inconsistency management requirements outlined in the previous section.

Database views and active constraint triggers can be used to build multiple-view systems where the views are informed of changes to model objects and requery the model to update the view’s state [53, 1]. For example, MELD [42] and MultiView [1] use a form of database views to support multiple views for software tools. Uni-directional constraint systems, such as Garnet [54], Clock [25], Zeus [12], and those built on top of database management systems, use constraint rules between software specification components which, when triggered, automatically update affected structures or flag the presence of inconsistencies. Multi-directional constraint systems, such as Rendezvous’s Abstraction-Link-View [38] and Amulet [55], use more flexible inter-object constraints allowing changes made to repository or view specification objects to maintain view consistency.

All of these trigger and constraint-based approaches use logical constraints and queries over model and view objects, and are thus able to readily detect structural and semantic inconsistencies. They do not, however, represent resulting inconsistencies without using objects or relations to model them, nor are they able to readily associate extra information with inconsistencies. For example, FormsVBT [6], built with Zeus, supports multiple textual and graphical views of user interface specifications and simulations, which are kept consistent under change. While FormsVBT successfully manages to keep views consistent, the

authors admit their techniques used do not scale up to more general software specification views, and lack more general inconsistency management facilities [6].

Smalltalk Model-View-Controller [46] and Java-style Observer [24] models support the notion of views of data structure objects, with the ability to propagate objects describing model object changes to observing objects. These “change” objects can be used to represent a range of inconsistencies, and extra information about inconsistencies. However, the architectures lack any application-independent approaches to generating such inconsistency representation objects, and many systems using these architectures simply rely on view objects to reconcile their state to their model objects i.e. using model change notifications as simple event triggers. This severely limits the kinds of inconsistency management that tools can provide to software developers. The ItemList structure [14] and Object Dependency Graphs [74] both use objects to represent structure changes in their inter-view propagation mechanisms. However, these objects are only used to reconcile viewing object structures to their models, and to implement undo/redo and hierarchical change propagation mechanisms. They do not support presentation or monitoring of longer-term inconsistencies. The View Mapping Language (VML) [2], which primarily uses constraints to map changes between schema views, could be extended to provide some of these capabilities, with objects representing inter-view inconsistencies.

FIELD [63, 64], and its successors, such as DECFUSE [37], use selective broadcasting to propagate messages about tool events between multiple Unix tools. Limited forms of view consistency are supported by FIELD, and building such environments and integrating new tools into the environment requires much effort [53]. CSCW toolkits, such as Groupkit [66], which uses a similar approach to FIELD for informing multiple users’ views of changes, also lack the range of inconsistency recording and presentation facilities required. CORBA-based multiple-view tools, such as those proposed by Emmerich [16, 18], may provide appropriate capabilities, by using combinations of remote object change broadcasting and shared data access for integrated tools. GSTL [17, 19] supports the generation of incremental consistency evaluation algorithms for tools by utilising document schemas and semantic relationship specification. GSTL also supports versioned documents and concurrent tool interactions, allowing tools to support differings levels of granularity and user coupling for cooperative editing. While GSTL-generated environments, such as an integrated SEE for British Aerospace [5], support powerful cooperative work, configuration management and inter-document dependency management facilities, they lack the range of short- and long-term inconsistency presentation, monitoring and interaction capabilities of tools like SPE. A few systems directly provide inconsistency management support, where multiple views need to be inconsistent for some time, but this inconsistency needs to be recorded and resolved at a later date. An example is [21], which uses logic predicates to record inconsistencies between different viewpoints on software designs.

Some software development tools, such as Mjølner environments [49], take the approach of preventing tool users from representing software artefacts in multiple views at all, therefore theoretically limiting possible inconsistency management problems. However descendants of Mjølner-based systems for multiple-user software development [50] have introduced simple inconsistency presentation and highlighting techniques to support version control and multi-user editing. PECAN [61], Garden [62], and Dora [59] all support multiple views of software artefacts, kept consistent via MVC-style object observation or database triggers. While a rich range of views is supported by these environments, they provide limited support for managing long-term inconsistencies between views. They also rely on structure-oriented editing techniques in order to keep views consistent, which lack favour with software developers [4, 73]. The Cornell Program Synthesizer [65] uses constraint expressions to present semantic inconsistencies to developers in views of software artefacts, which can potentially exist for long periods, but these can not be interacted with nor be grouped.

Most CASE environments use the notion of a repository, with database view mechanisms to keep multiple views of artefacts structurally consistent, and triggers and queries to detect semantic inconsistencies [23]. For example, Software thru Pictures™ [72, 40], uses a Sybase database, EiffelCASE [41], uses persistent Eiffel objects, and Rational Rose [60], uses files. Inconsistencies detected by constraint triggers are acted on immediately and queried inconsistencies may lack information about the context (such as part of the software process) a change was made in, who made the change, when it was made, and why it was made. As inconsistencies detected by triggers or querying are generally not represented and stored in the tool repository, inconsistency representations can not be annotated with additional information about them at the time they are generated and stored for monitoring.

CASE tools generally use reverse engineering and merging tools to reconcile modified CASE and code views [41, 40]. They often require separate configuration management and shared repository tools to enable collaborative development. For example, Rational Rose uses a separate product, ClearCASE, to maintain shared CASE documents via a configuration management tool with checkin/checkout policies [60]. A

similar approach is taken in EiffelCASE [41,] and Software thru Pictures [40]. This approach delays the detection of inconsistencies between design and code views and thus fails to provide immediate feedback to developers when such inconsistencies occur. This often leads to many inconsistencies between design and implementation views of a system that require a large effort to resolve later. Object Team [13] utilises a repository with built-in configuration management, continually synchronising design and code artefacts. This approach leads to intolerance of partial inconsistency between developers, which can greatly hinder useful parallel design and development exploration [21]. The increasing need for a range of multi-user support facilities in CASE tools [45] also means that architectural approaches which better facilitate the management of multi-tool and multi-person inconsistencies, such as selective broadcasting, CORBA-style remote notification and process-centred support, need to be used.

Meta-CASE tools usually provide some degree of support for generating multiple-view supporting CASE tools. MetaEDIT+ [43] and MultiView [1] provide limited view consistency management, based on database views. MetaView [67] provides view inconsistency management using constraints, with basic facilities to allow developers to monitor, group or interact with inconsistencies. They do not support the annotation of inconsistency information with context that can aid the user in understanding the reasons for a change. KOGGE [15] generated tools use a database with active object views to keep multiple viewpoints consistent at all times, with no tolerance for inconsistency.

Process-centred environments allow software developers to plan and coordinate their use of multiple tools and work on multi-user projects. Oz [11] and Merlin [57] use rule-based approaches to describing process models, and Oz utilises “enveloping” to integrate other tools with the process-centred environment [70], but do not provide management of inconsistency representations generated by integrated tools [51]. TeamWARE Flow [69], Action Workflow [52] and Regatta [68] use more developer-accessible, graphical process modelling languages, and provide simple interfaces to coordinating third-party software development tools, but do not manage inconsistencies. SPADE [8], ProcessWEAVER [20] and ADELE/TEMPO [10] provide more sophisticated facilities for integrating third-party tools, with some inconsistency management, event handling and constraints applicable to integrated tools. However, the degree of integration means only basic inconsistency monitoring can be facilitated, and inconsistency presentation and interaction within integrated tools is not supported [9]. The process modelling and enactment approach of [47] has been used to coordinate multiple view usage [58], allowing developers to specify how inconsistencies detected in multiple view tools can be handled.

4. Inconsistency Representation using CPRGs

The first four inconsistency management requirements we identified in Section 2.2. relate to the ability of software development tools to adequately represent software specifications and to detect and represent inconsistencies. As existing software architectures do not support all of these inconsistency management requirements, we have developed a new software architecture, Change Propagation and Response Graphs (CPRGs), for managing inconsistencies in multiple view software development tools [29]. We first motivate the need for this architecture and explain its design rationale, with a simple example of its use for inconsistency management. We then describe the realisation of this architecture in tools which support the construction of CPRG-based environments.

4.1. A Model for Inconsistency Management

Software specifications consist of a wide variety of “software artefacts” i.e. different kinds of information about software which together comprise a full (or partial) system specification at various levels of abstraction. Some of these artefacts are structured, textual or graphical documents, while others are more loosely structured. Graph-based structures tend to suit the representation of fine-grained software artefacts, such as abstract syntax trees and graphs for diagrams and code [4, 65, 7]. However, in order to produce efficient environments which can make use of existing tools, such as text editors, coarse-grained representations of parts of software specifications, such as the code associated with a class method, can also be represented as a single “artefact” [27, 59].

The CPRG software architecture is based on a graph-based representation for software artefacts, with attributed *components* linked by inter-component *relationships*. CPRG components can represent small, fine-grained software artefacts, such as classes, attributes, methods, etc. They can also represent coarser-grained artefacts, such as entire class interfaces, third-party tool documents, such as MS Word™ files, and interfaces to third-party tools and databases. Multiple views of software specifications are built with CPRGs using “view components”, with these view components related to “base” (i.e. repository) components via “view relationships”. This is illustrated in Figure 2 with a class method shown in two

different views (one a graphical design view, the other a textual code view). The top windows are graphical and textual views with which the user interacts. The middle layer is a CPRG representation of these views, and the bottom layer is the repository of the environment containing a CPRG describing all software specification information.

Changes to graph structures representing parts of a software specification lead to inconsistencies when: i) related structures that share updated information are not changed, e.g. multiple views of a changed component are not appropriately updated; ii) semantic constraints on components are violated e.g. a class trying to use a method in another class uses the wrong number or type of arguments; and iii) specifications updated by one developer become inconsistent with those of other developers. CPRG components and relationships are used to embody both the structure of software specifications and the semantic constraints within and between parts of specifications. When these detect that related components have changed, they carry out structural or constraint checks for inconsistency. Some CPRG components may embody only structure or only semantic constraint information, but often many embody both. When developing CPRGs, we chose not to introduce specialised kinds of relationships for e.g. structural relationships versus semantic constraints, in order to keep the model simple but also to ensure a homogeneous approach to handling both structural and semantic inconsistencies.

The state of a CPRG component is modified by an *operation*. When an operation changes the state of a component, objects describing this state change, called *change descriptions*, are generated. Change descriptions generated by a component are propagated to all connected CPRG relationships, which then decide to forward the change to other components, act on the change, or ignore it. If related structures are not updated to reflect the changed state of the modified component, change descriptions indicate structural inconsistencies. Change descriptions are also used to represent semantic inconsistencies, by having change descriptions generated when constraint violations are detected. Components generating or receiving change descriptions can also annotate them with extra information, such as the time and date the inconsistency occurred, the developer who caused it, what process model stage was enacted when it occurred, the relative importance of the inconsistency, or any other additional reason for this inconsistency being represented. Change descriptions can be grouped with components to “record” the presence of inconsistencies associated with these components. These groups of change descriptions can be browsed by users and searched and acted on by software development tools. For example, the history components in Figure 2 are used to record the modification history of views and repository-level classes.

An additional reason for using change description objects, rather than constraints or traditional model-view change notification, is the homogeneous solution change descriptions provide for inconsistency management, tracking and versioning component changes, and collaboration support [29, 30]. Change description storage supports the formation of “modification histories” which track changes that components have undergone. In the Serendipity process modelling environment we also associate groups of change descriptions from software artefacts with enacted process stages, showing the history of work done for each stage when it was enacted [34]. CPRG change descriptions representing state changes can also be reversed to “undo” these state changes, or re-applied to repeat the state changes. This combination of recording changes in groups and undo/redo supports “deltas” for each component, supporting a basic versioning mechanism. Change description objects can also be serialised and broadcast between users’ CPRG-based environments, facilitating a range of collaborative work facilities [31].

Figure 2 illustrates an example use of CPRG change descriptions to support inconsistency management in SPE. The name of a method in a class diagram is modified by the user (1), resulting in a change description being generated. This change description is propagated to the view and stored in the view’s modification history, and propagated to the method view components’s view relationship, which automatically updates the repository method component (2) to remove the structural inconsistency between repository and view. A change description resulting from this update of the base method component is generated (3), and then propagated to all related components interested in changes to the base method (4). If guiding software development in SPE with a Serendipity process model, the change description will also be annotated by Serendipity with information about the developer and enacted process stage. The view relationship informs all other views of this base component of its state change (5), and views are either fully or partially updated or inconsistencies recorded and presented to the user, with affected view components highlighted. Classes which use this method check semantic constraints on the usage (6). If the change to the method causes their use of it to become invalid e.g. their code calls the wrong name, or wrong arguments, a semantic change description is generated. If the user has specified this kind of inconsistency is important, the change description will be annotated with e.g. a “medium” importance indicating the developer must correct the change before the system can be compiled, or a “high” importance indicating the inconsistency should be resolved very soon. The owning class of the updated method is informed of its state change, and records the change description in its modification history (7), tracking all changes to the class. The original change

description generated by the view component update can be broadcast to other developers sharing the edited view, to support a variety of collaborative editing and version merging facilities.

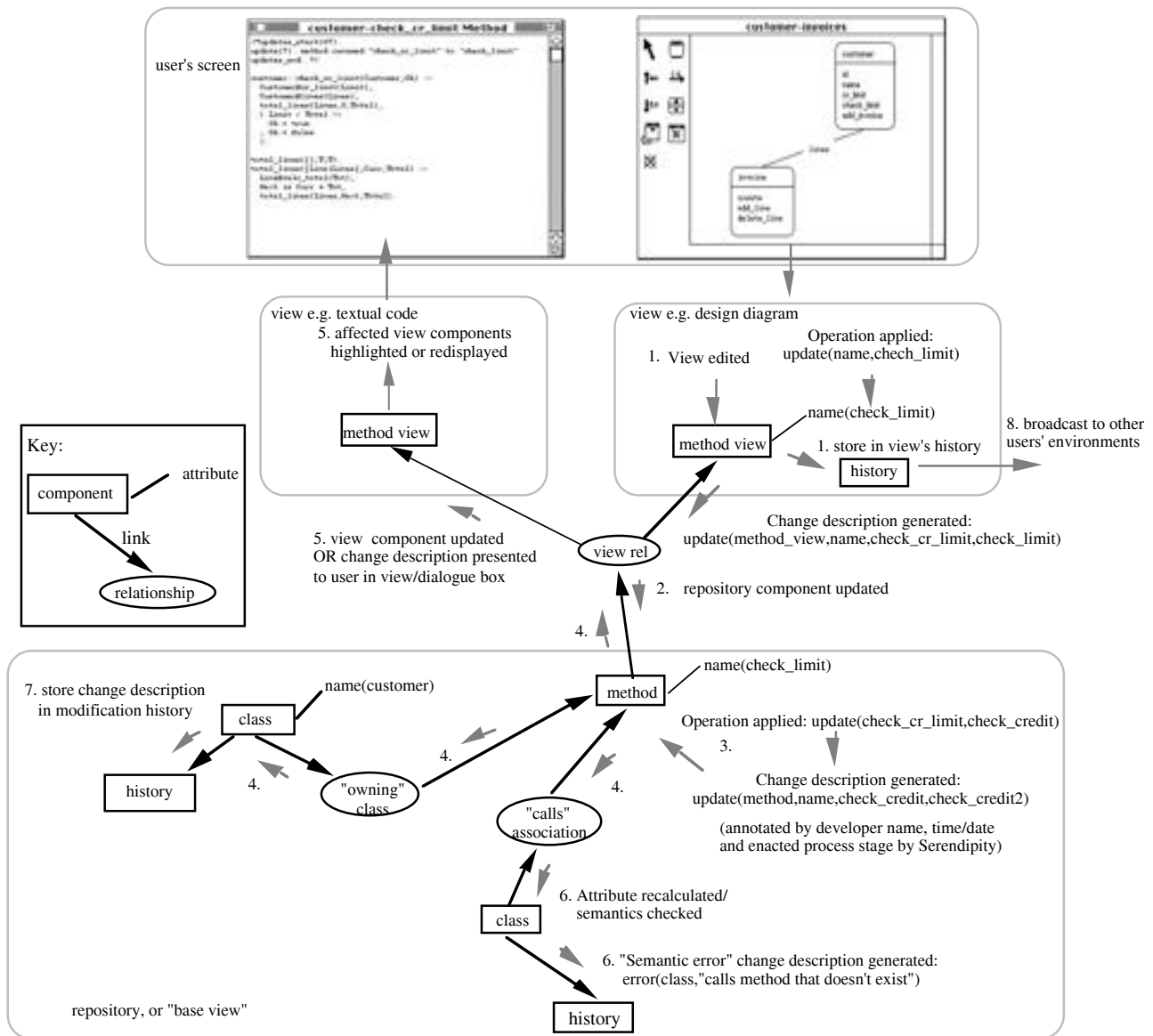


Figure 2. An example of inconsistency management using CPRGs.

4.2. A Software Architecture and Support Tools for Realising the CPRG Model

We have developed object-oriented class frameworks and an environment generator to allow software tool developers to use the CPRG architecture to develop multi-view, multi-user environments. We have used these frameworks and generator to construct several exemplar development tools and environments exhibiting a range of inconsistency management facilities.

4.2.1. MViews and Serendipity

Our first realisation of CPRGs was the MViews class framework for building multi-view editing environments [30]. We chose an OO framework approach to enable tool developers to reuse the basic CPRG functionality via inheritance and composition of CPRG-implementing classes, and then extend this basic functionality for use in their own tools by writing additional, application-specific code. We built a range of specialised classes for MViews that extend the basic CPRG model. These provide abstractions for building multiple view representations of software artefacts, a variety of inter-component relationships, a variety of graphical and textual editor building-blocks, and support for collaborative work in MViews-based tools.

We implemented MViews in Snart, an object-oriented Prolog extension [26]. We chose to represent change descriptions in our Snart implementation of MViews as Prolog terms, rather than Snart objects, for speed of generation, propagation and storage. Change description terms in our MViews implementation represent CPRG component state changes and constraint violations, with a Prolog list appended to enable the annotation of change descriptions with additional information about inconsistencies. Collaborative view editing and version sharing is supported by broadcasting serialised terms between users' environments and annotating change descriptions with user names. Snart is a persistent language, with objects dynamically saved and loaded to a persistent object store, making information repository and view persistency management transparent for environment implementers.

In MViews-based environments, such as SPE, it became apparent that users require additional "work context" information about inconsistencies, and for inconsistencies to be grouped not only by affected artefacts but also by user and the context in which the inconsistency arose. We chose to "capture" additional reasons about inconsistencies (and software artefact modifications in general) by using software process model information to annotate MViews change descriptions with user, time and date, process model stage, and reason for stage enactment information. This led to the development of the Serendipity process modelling environment, and integration of Serendipity and MViews-based environments [31, 34]. We modified MViews so that any change descriptions generated by view components or base layer components are forwarded to Serendipity (if in use), and these change descriptions are annotated with enacted process model information. They are also copied and stored against the enacted process model stage to facilitate grouping of inconsistency representations and modification histories with process model stages in Serendipity. Figure 1 shows examples of annotated change descriptions in SPE, and also stored change descriptions originating from SPE modifications stored against a Serendipity process model stage.

4.2.2. JViews, JComposer and Serendipity-II

We built many multiple-view, multi-user software development environments with MViews. However, MViews-based environments suffer from slow performance due to the Prolog implementation of MViews, difficulties in integrating third-party tools not built with MViews, and a large amount of effort required by developers to implement MViews-based tools. This led to the development of JViews, a Java-based implementation of CPRGs, which uses and extends the Java Beans change notification mechanism to represent change descriptions and support change description propagation [33]. JViews is an object-oriented class framework with similar capabilities to MViews, but its architecture is much more open for tool development and integration. JViews uses the Java Beans componentware API to allow events from third party tools to be represented and used within JViews environments, and to be able to send instructions to third party tools which provide component-based interfaces. A variety of components supporting collaborative view editing, component persistency and repository management, and software artefact representation are provided by JViews. JViews does not currently use a common architecture for distributed object management, such as CORBA. However, we are investigating use of the remote object change notification mechanism in CORBA for representing change descriptions and their propagation for JViews-based environments.

To reduce development effort with JViews, one of our first JViews-based environments was JComposer, a tool for specifying CPRG-based environments and generating JViews-based implementations [33, 35]. We decided to generate JViews implementations of tools rather than interpret JComposer specifications to ensure efficient, stand-alone and interoperable tools would result. Figure 3 shows an example of part of the specification of the Serendipity-II process modelling tool, a reimplementing of Serendipity using JComposer and JViews. The JComposer view on the left shows part of a specification (which uses the CPRG notation) being developed. A view from the generated environment running is shown on the right. Serendipity-II can be used to provide a context for inconsistencies in JViews-based environments, in the same manner as Serendipity does for MViews-based environments.

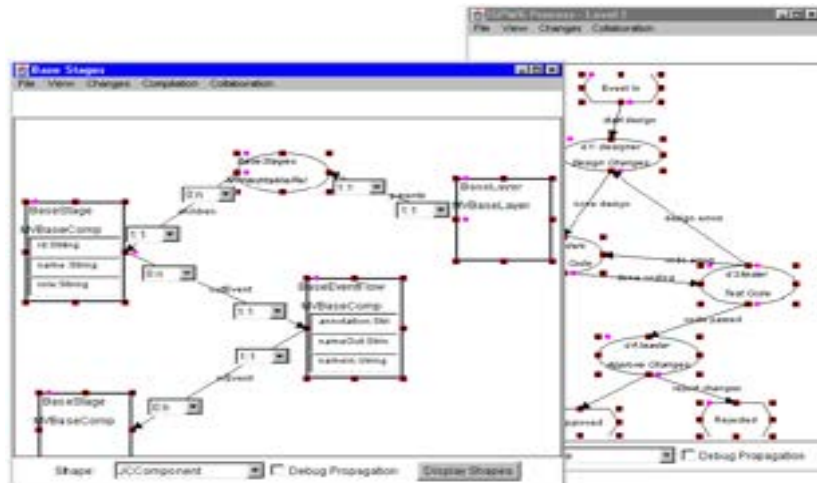


Figure 3. An example of the JComposer tool and the generated Serendipity-II in use.

5. Querying and Presenting Inconsistencies

Any environment supporting multiple views needs to either automatically resolve view inconsistencies or to inform developers of unresolved inconsistencies [53, 61, 63]. Often inconsistencies between multiple views can be automatically resolved by modifying the state of one or more views to reconcile them to the modified state of other views. An initial consideration is whether a change description can and should be automatically applied to a view receiving the change to rectify the inconsistency it represents. An example where this is possible is the renaming of a class in an SPE class diagram, which is easy to propagate to and automatically effect in other class diagrams showing the class. CPRG-based environments readily support this behaviour by having response methods in the components that receive change descriptions pattern match against received changes and perform actions to rectify inconsistencies. SPE uses this technique extensively for maintaining graphical view consistency. Many existing multiple-view software development systems support only this level of view consistency management. If a change can be directly made to another affected view, it is performed by the environment. If it cannot, it is ignored and the user of the environment is usually never informed of a potential inconsistency.

Just because an inconsistency can be automatically rectified it is not necessarily the case that it should be. For example, a far-reaching change made by an inexperienced colleague may be better handled as an inconsistency that is presented to other developers. In addition it is often important to track that an inconsistency occurred and that it was automatically resolved by the environment. Thus in SPE we have chosen to have all changes to a view or a component recorded in modification history lists, able to be reviewed using the techniques described in the following sections. For any important changes which are automatically resolved by an environment, it is also often useful to highlight the change *after* it has been made, to draw developers' attention to it. For example, in SPE changes such as renaming classes and methods may be partially automatically resolved between views, but the changed items in views are highlighted.

5.1. Highlighting the Presence of Inconsistencies

CPRG-based environments inform components of the possible presence of inconsistencies by propagating change descriptions to them, and record the presence of such inconsistencies by associating change descriptions with them. These environments must then inform developers that such inconsistencies have been detected, either in a "context-dependent way" (e.g. indicating what they affect in the views with which developers are interacting), or in a "context-independent way" (e.g. by grouping and presenting related inconsistencies together). Highlighting parts of a view supports a basic "context-dependent" approach to informing developers of inconsistencies which affect the view.

A variety of techniques can be used to highlight the presence of inconsistencies, including shading and colouring graphical icons, changing text font characteristics, or blinking affected icons. Annotating the name of a view's window, or the values of text displayed in the view are also often appropriate and easily implemented. Many of these techniques are equally applicable to textual and graphical views, and many can be usefully combined. The choice of technique often depends on how developers can most appropriately be informed of the presence of different kinds of inconsistencies. For example, inconsistencies which need

quick resolution to allow software development to proceed should be immediately presented so that developers readily notice and act upon them. Inconsistencies which can be tolerated for longer periods, or which do not have a large impact on the information displayed in a view, are usually more effectively presented in less dramatic ways, or may even be highlighted only when requested by developers.

As a simple example, consider the screen dump from the dialogue definer of SPE in Figure 4. This shows three views of a dialogue under design, a graphical drag-and-drop composition (dialog1-dialogue), a textual specification (dialog1-Dialog Predicate) and an example of the running dialogue. In this example, the ‘Ok’ control button in the graphical view has been shifted, resulting in a semantic constraint that dialogue components not overlap the dialogue border being violated. A change description has been generated representing this inconsistency and associated with the dialogue view’s Ok button component. Such an inconsistency must be resolved before the dialogue specification can be used. Thus SPE immediately indicates the presence of such semantic inconsistencies by shading the Ok button icon, to highlight it. This technique could be used in combination with other context-dependent inconsistency presentation approaches, such as boldening the textual Ok button specification in the right-hand view, renaming the views to e.g. “dialog1-dialogue (error)” to indicate the view has an inconsistency, and so on.

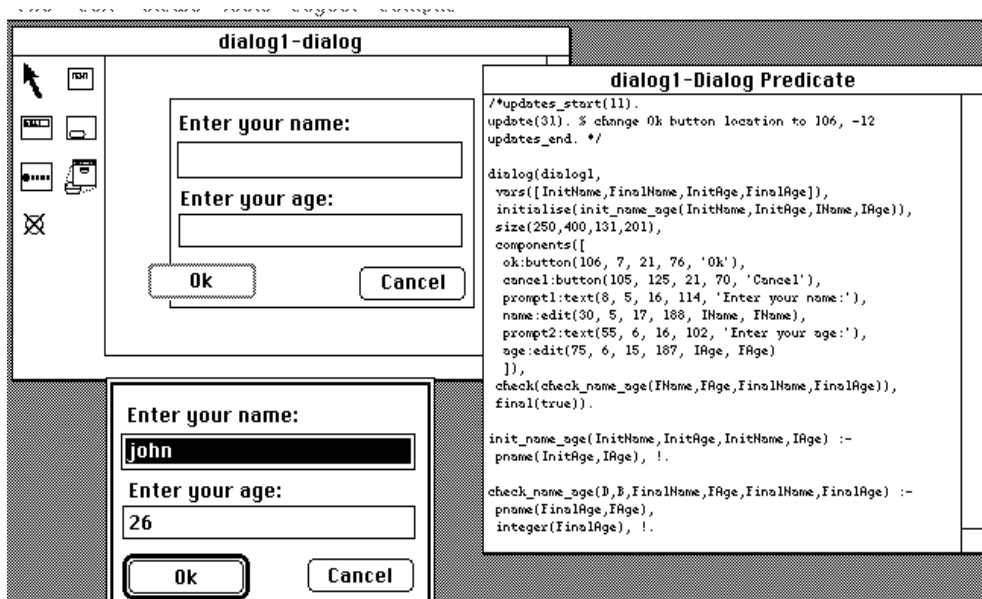


Figure 4. Indicating inconsistencies in graphical views.

5.2. Textual Presentation of Change Descriptions

While the presence of inconsistencies can be indicated in a variety of ways, developers often need more information about them. For example, a developer may not understand why the Ok button in Figure 4 has been highlighted and wants to view more information about the inconsistency this indicates. As described in Section 4, change descriptions may embody a wide variety of information about an inconsistency. Textual forms of change descriptions can be presented to developers by inserting them into views or displaying them in pop-up menus and dialogue boxes. An important characteristic of all CPRG change descriptions is that the inconsistency information they embody can be “unparsed” and viewed in a textual, human-readable form.

For example, Figure 5 shows the textual forms of several change descriptions from SPE indicating: 1) the addition of a method to a class in SPE; 2) a semantic error denoting an unknown method is being called; and 3) an annotated analysis change. For change description #1, additional information from Serendipity has been used to annotate this particular change description, i.e. the process stage (*aff.2.1:Design Changes*) and developer making the change (*judy*). This annotation allows users to determine the reasons the change was made or inconsistency detected, who caused the inconsistency, and so on. For change description #2, the “importance” of the semantic inconsistency is indicated with the three stars prefixing the description. For change #3, an additional annotation indicating the change was made in an analysis diagram has been added, indicating that when this change description presentation is viewed for design and/or code views, the change needs to be appropriately propagated to specifications at these levels of abstraction.

1. [aff.2.1:Design Changes-judy] add feature customer::calc_fines

```
2. *** Unknown method 'check_limit' being called in customer::check_cr_limit
3. [*** analysis] change association [customer::acc-of, ->, account] arity
from 1:1 to 1:n
```

Figure 5. Textual form of a change description

5.3. Grouping of and Querying for Inconsistencies

Change descriptions can be grouped and associated with various CPRG components they affect. Such grouped change descriptions provide an underlying "database" for querying about inconsistencies. By applying selection operations on this database, collections of change descriptions relating to a particular component, view, process stage, or inconsistency type can be constructed, and then appropriately presented to the user.

For example, SPE keeps histories of modifications and semantic constraint violations for all class components and views, but also records groups of all constraint violations, unresolved "inter-specification" changes (e.g. design changes affecting code and vice-versa), inconsistencies resulting from version merging, and changes by Serendipity process stage. These groups of stored changes are then used to create a variety of change and inconsistency presentation lists for users to interact with. Figure 1 shows two such lists, one for a Serendipity process stage, and one for an SPE artefact (class customer). Each change description in the lists is displayed in textual form, annotated by a sequence number indicating the order they were generated. Developers can also configure environments constructed with our CPRG-based tools to record inconsistencies in user-specified ways (see Section 8).

An example use of presenting grouped inconsistencies is shown in Figure 6. Changes made to an SPE class are shown, with some changes having been translated from changes made to a same-named EER entity [71]. Items in this change history list highlighted with a '*' were actually made in the EER view and translated into OOA/D view updates. The user can make further changes to the OOA/D view if the EER update could only partially be translated into an OOA/D view update i.e. an inter-notation translation inconsistency occurred. For example, adding an EER relationship (change #8) was translated into the addition of an OOA/D association relationship (change #9). The user then refined this relationship to an aggregation relationship (change #10). This could not automatically be done, as the EER notation does not support the distinction between different kinds of relationships. This technique of presenting inconsistencies between design notations in a textual list is combined in SPE with highlighting the presence of possible inter-notation inconsistencies by shading icons in a view affected by changes to another notation view [71].

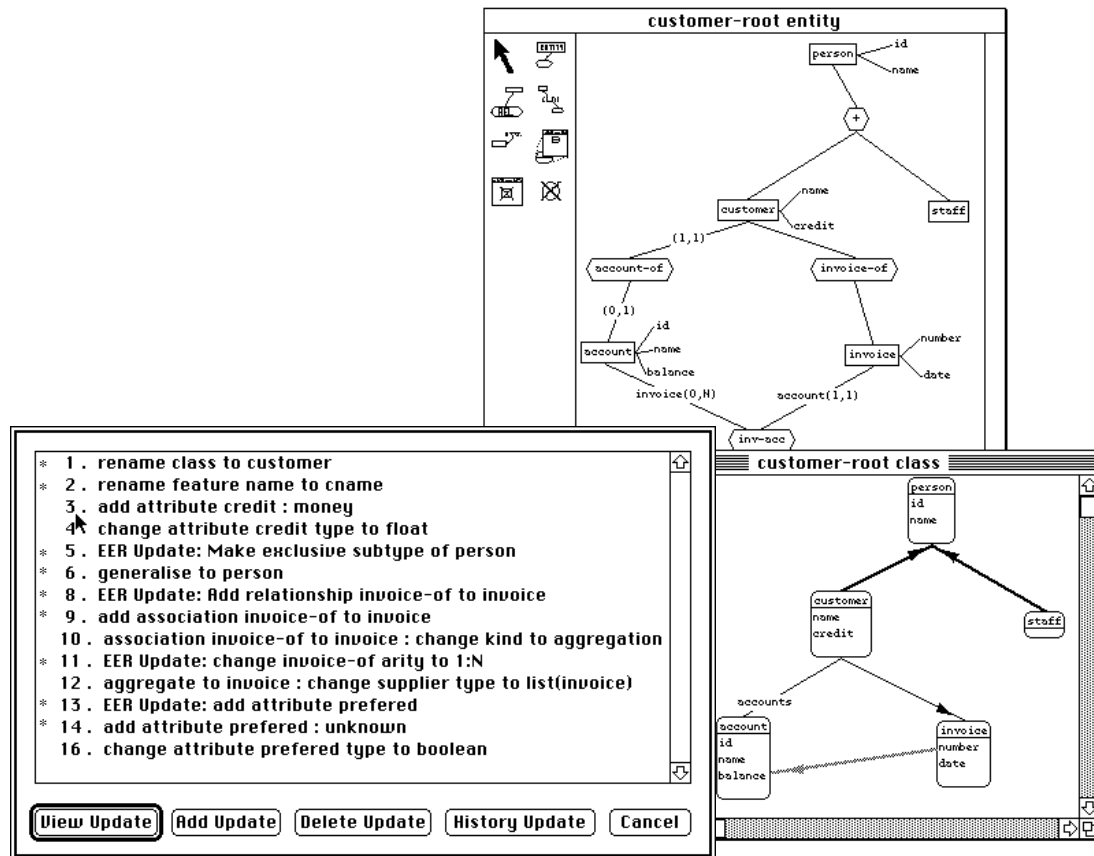


Figure 6. Graphical view inconsistencies.

5.4. Displaying Inconsistency Representations in Views

One application of grouped change descriptions commonly used by our tools is to present currently outstanding inconsistencies by annotating the contents of views. That is, rather than just highlighting view components to indicate the presence of inconsistencies, inconsistency descriptions are inserted into the view. This provides more immediate information to the user than simply indicating which parts of a view are inconsistent. We achieve this either by inserting textual descriptions of change descriptions directly into a textual view or by adding pop-up menu items to a view where such textual descriptions can be readily accessed. Care must be taken to ensure these descriptions are put into a logical place for developers to access, and to distinguish them from actual software artefact specifications. We have found specially distinguished comment areas for textual specifications and pop-up menu handles for graphical views most appropriate.

For example, Figure 7 shows an SPE class interface view with several change descriptions representations inserted into a special header region at the top of the view. These describe inconsistencies between this view and other, modified views of the program, and also semantic errors caused by constraint violations (e.g. same-named items). In this example, change 32 indicates the address attribute has been renamed to address1, change 33 indicates addition of attribute address2, change 36 indicates addition of a client/supplier relationship between the customer::add_invoice method and the invoice::create method, and change 44 indicates a compilation (semantic) error due to the duplicate calc_total_owed methods. We chose to present change descriptions at the beginning of SPE class and method implementation specifications, and when developers open such views these inconsistency presentation regions are updated with new inconsistencies inserted after older ones. We also allow developers to edit this text to remove inconsistencies they have actioned or do not want to continue to see in the view. However, such inconsistencies are kept and can still be queried for and viewed in a dialogue.

The first three of the presented inconsistencies in Figure 7 might have been made in a graphical view and thus the change descriptions inform the programmer of (possible) view inconsistencies between this textual view and the modified graphical view. Changes 32 and 33 can be automatically applied by SPE to the textual view. The developer can configure SPE to always automatically update the view's text to reflect such changes, rather than displaying the change descriptions. To resolve change 36, the programmer may,

at some later time, modify the customer::add_invoice method to insert an appropriate method call and arguments, and possibly update the customer class so that an appropriate reference to invoice class objects exists. It can be difficult to describe the full cause of semantic inconsistencies, such as Change 44. Often an environment may only inform users of the editing change(s) that caused the inconsistency and the semantic constraint violations detected.

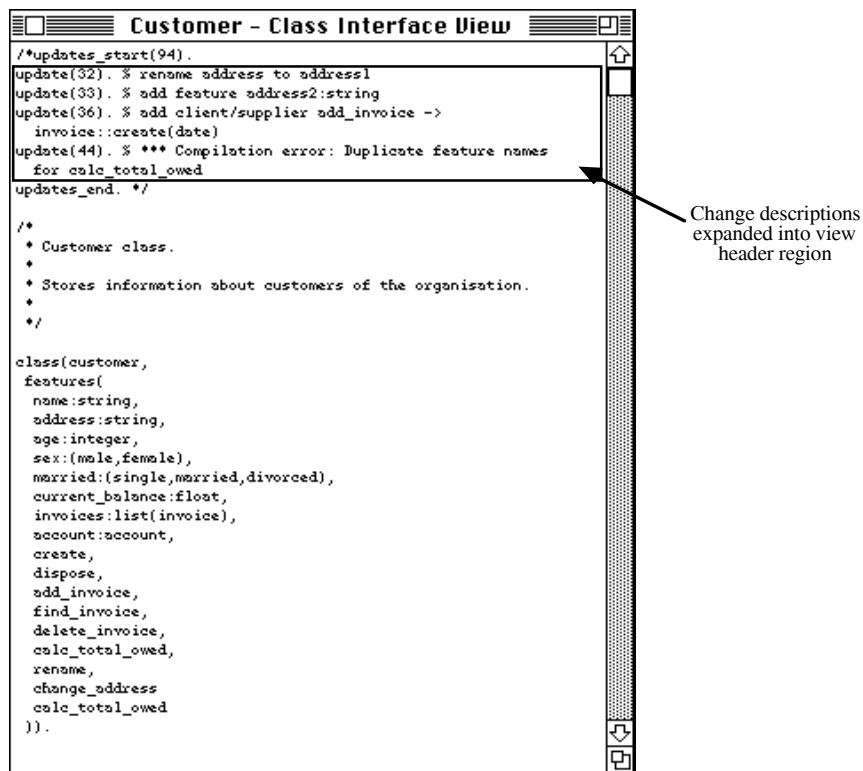


Figure 7. Indicating inconsistencies in textual views.

An interesting use of the above technique in SPE is its application to documentation views. SPE allows artefacts, such as classes, to have associated documentation views describing them. Change descriptions affecting the artefact are forwarded to the documentation views and automatically inserted into the view header, acting as an indicator that the documentation may require updating to reflect the change.

6. Interaction with and Resolution of Inconsistencies

Once developers are aware of the presence of inconsistencies, they may wish to interact with inconsistency presentations or indeed resolve these inconsistencies. Interaction may simply be clicking on an inconsistency presentation to gain more information about it. For example, clicking on the shaded Ok button from Figure 4 to be shown a detailed description of the inconsistency, like those in Figure 5, in a dialogue. Interaction may also include selecting inconsistency presentations and requesting the environment to: resolve them automatically; mark them as “seen”; change the importance or other information associated with them; move them to a history list for later action; or delete them (i.e. the inconsistencies are tolerated). Because inconsistency presentations in our systems are representations of CPRG change description objects, providing developers with the ability to interact with inconsistency presentations can be seen as providing facilities to manipulate these change descriptions. This section illustrates some of these techniques as used in SPE, and the rationale for their choice in different situations where developers need to interact with inconsistency presentations.

6.1. Selection and Resolution of Inconsistencies

Many structural inconsistencies between views can be automatically resolved by our environments. For example, for textual and graphical views SPE can automatically expand added or hide deleted classes, methods, method arguments, local variables and attributes, and can automatically rename classes, methods and attributes, and change attribute and method argument names and types. In graphical views, SPE can automatically expand added, change modified and hide deleted classes, class components and inter-class analysis and design relationships [27]. When structural inconsistency resolution actions are automatically

carried out, changed view items are highlighted to indicate they have been modified and the actioned change descriptions stored in a history list associated with the view for developer perusal. When such structural changes cannot be automatically resolved the inconsistencies are presented as described in Section 5.

Developers may choose to have inconsistency presentations shown in a view or associated history list dialogue, and not be automatically resolved by an environment. For example, with SPE we have found developers want automatic resolution when using graphical views, but with textual views often want all inconsistencies shown in the textual view header, whether they can be automatically resolved by SPE or not. This is because even when apparently simple graphical view analysis and design changes are made, such as renaming attributes etc. which can be automatically applied to some affected textual implementation code, developers like to be informed of such inconsistencies and determine when they are resolved. They also like to have both automatically resolveable and non-automatically resolveable inconsistencies handled in the same manner for textual implementation views, but prefer automatically resolveable analysis and design changes to be applied to graphical views when they are detected by SPE. This behaviour can be changed by developers if required, as described in Section 8.

For inconsistencies that can be automatically resolved, the user can select one or more inconsistency presentations in a dialogue or view, issue a request to implement the change, typically via a pull-down or pop-up menu item, and are informed of the results. Figure 8 shows an example of such developer-requested automatic view inconsistency resolution in SPE. The user has selected all of the change descriptions in the view header and asked SPE to update the view's text. The first two changes can be automatically applied by the environment and result in attribute address being renamed to address1, and addition of attribute address2. Both change descriptions are deleted indicating successful update of the view. The other two changes, however, cannot be automatically applied, and the change descriptions are left in the view's text to indicate this. Updates selected in a dialogue but which could not be successfully applied by the environment are highlighted or shown to the user in another dialogue.

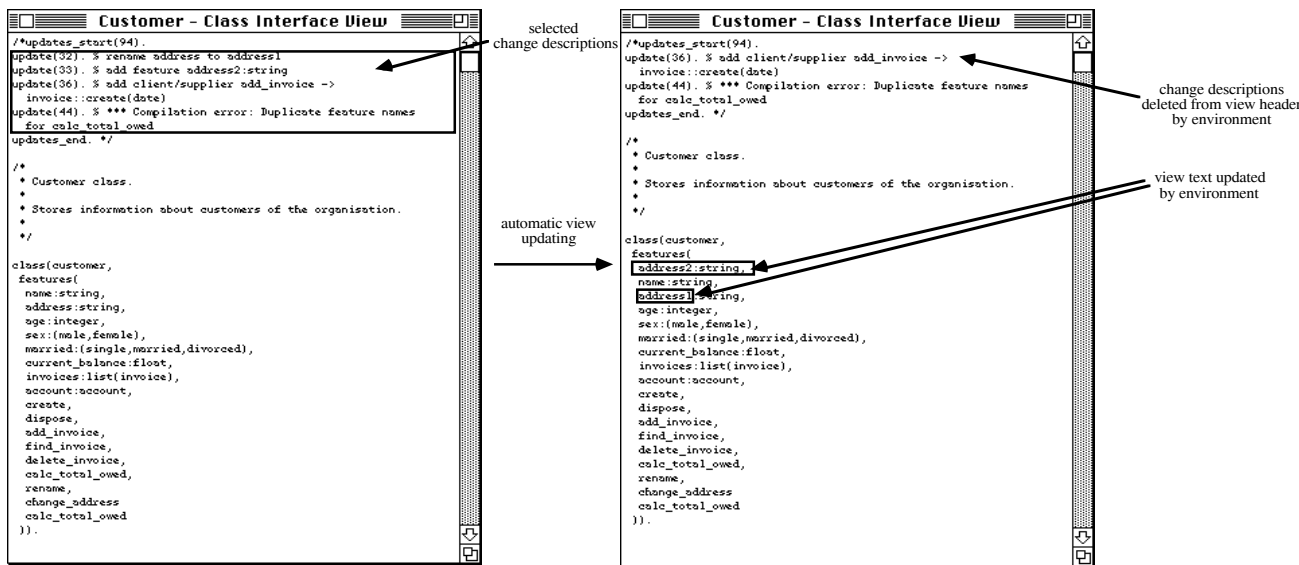


Figure 8. Automatic resolution of view inconsistency under developer control.

6.2. Selection of Optional Update

Often environments can partially resolve an inconsistency, or may determine several possible alternative approaches to resolving it. Environments in which such partial automatic inconsistency resolution is possible include those which integrate multiple design notations, such as OO, ER and NIAM models [71], multiple levels of specifications, as in SPE [27], and implementation, documentation and formal specifications [30]. Often a change made to one notation view can be partially translated into a change in views using another notation. The user may be able to complete the translation by choosing from a range of possible view changes.

An environment can assist in determining a partial inconsistency resolution or a range of alternative strategies. The environment can then make the partial change and leave the developer to complete the resolution, or facilitate the developer in choosing their preferred inconsistency resolution strategy.

Developers should be informed that a partial change has been made or that they need to complete a change, using similar highlighting and presentation techniques to those described in Section 5. Developers can choose an appropriate resolution strategy, for example, via a pop-up menu with resolution choices or by having several alternative change descriptions presented which describe each resolution and allowing developers to select the one they want.

Consider SPE facilitating the translation of changes between OOA and EER notations when a relationship is added in an EER view. For example, an EER relationship is added between customer and invoice entities, and SPE translates this into the addition of a relationship between customer and invoice classes in OOA/D views. However, more information is needed in the OOA/D views: the relationship might be an association or aggregation OOA relationship (not specified in the EER view), or if it is an OOD client/supplier relationship, it may specify a method call between objects and thus caller/called method names and arguments must be specified. As this information is not specified in EER views, we chose to have SPE default the relationship to an OOA association relationship, and let the user refine this further by changing it to an aggregation or client/supplier relationship by adding extra information. SPE indicates the new relationship has been defaulted by highlighting it, and provides a pop-up menu to allow the user to easily change its type and add appropriate extra information. Figure 9 shows an example of such an interaction with an inconsistency via optional update.

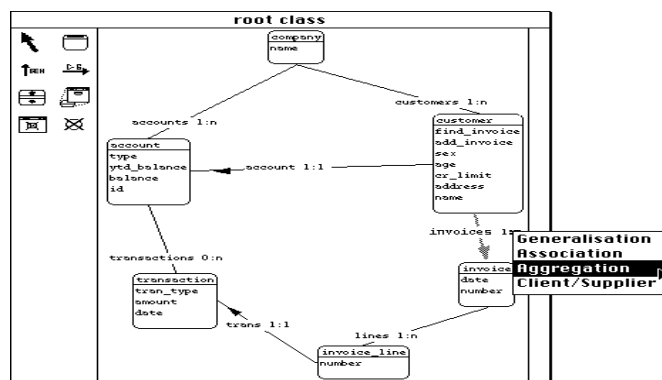


Figure 9. Selection of optional update to fully resolve an inconsistency.

6.3. Manual Resolution of Inconsistency

Some inconsistencies between views can be presented to developers but not resolved in any way by an environment, although the environment may suggest different ways the developer might manually resolve them. In this situation, the developer must manually resolve the inconsistency (which can be quite difficult and time-consuming). Using different highlighting techniques, our environments inform developers of the presence of such inconsistencies and may indicate they can not be resolved by the environment. Developers then resolve the inconsistency manually, indicating they have resolved it by deleting the inconsistency presentation, moving it to another history list, or associating information with it indicating its resolution (or partial resolution).

For example, the addition of a design-level client/supplier relationship in SPE is usually implemented as a code-level method call. Such a change cannot be automatically implemented in a code-level textual view, as the environment does not know the appropriate method arguments (variables or constants) and position of the method call within the method code. Similarly, semantic errors such as duplicate method or attribute names, type mis-matches and the calling of non-existent methods in other classes all require developers to take some appropriate action to correct the problem. Figure 10 shows an example of this manual inconsistency resolution to resolve a simple semantic inconsistency in a class definition view.

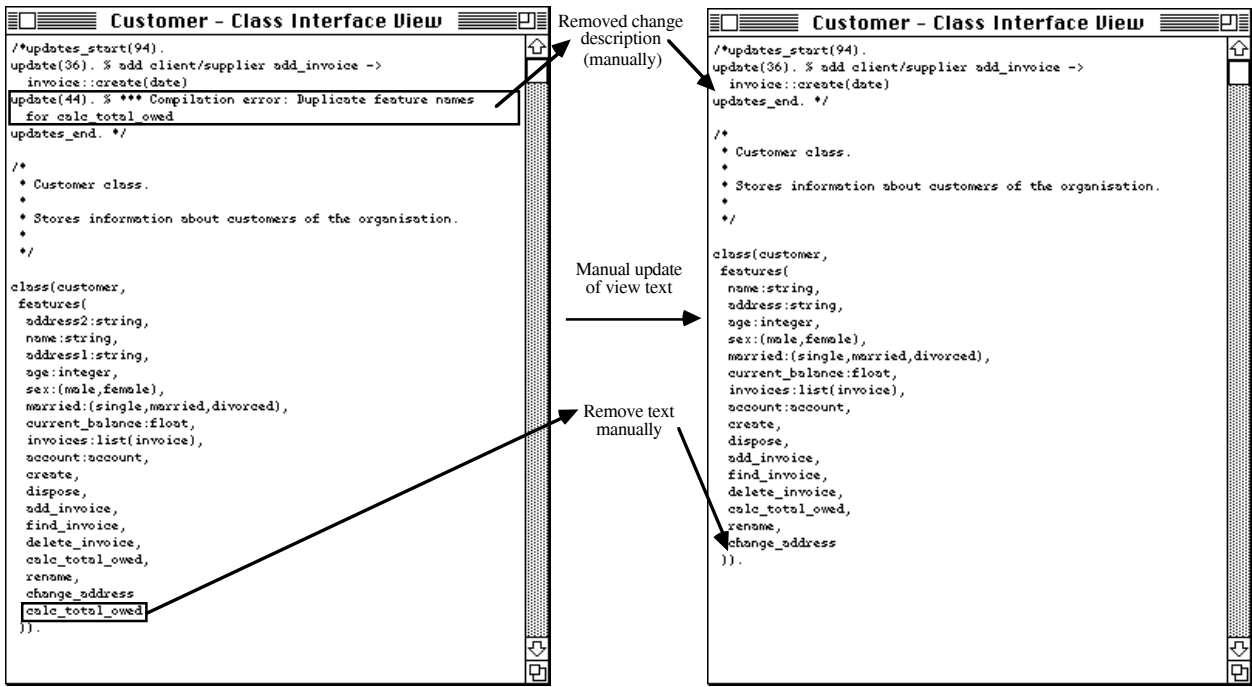


Figure 10. Manual resolution of inconsistency.

6.4. Other Interactions with Inconsistencies

In addition to resolving inconsistencies, users can interact with inconsistency representations in other ways. An environment can add interaction “hot-spots” to highlighted inconsistencies, providing access to a more complete description of the change in a dialogue or pop-up menu, or display the view whose update caused the inconsistency. Developers can also be provided with a mechanism to annotate an inconsistency themselves, to move it or copy it to other inconsistency recording groups, or to delete it (i.e. tolerate the inconsistency). We have found that providing inconsistency presentation selection facilities in views and dialogues, or even pop-up menus associated with inconsistency presentations, effectively supports such developer and inconsistency interactions.

For example, Figure 11 shows an SPE change history where the developer can select inconsistencies and ask for their affect to be undone/repeated (if structural changes), ask for more detailed information to be displayed, annotate the inconsistency, or delete it from the list. Developers may even create their own “user defined” inconsistencies and add them to the list. These do not necessarily represent a view inconsistency, but rather serve as user-defined documentation, perhaps describing the effect of a group of lower level changes which have been made. Such user defined change descriptions are associated with specific software components, and, like change descriptions representing structural and semantic inconsistencies are propagated to other views of these components. They also serve a useful purpose in supporting context-dependent communication in cooperative environments, allowing cooperating users to interact by “messages” sent via the change description mechanism [27, 29].

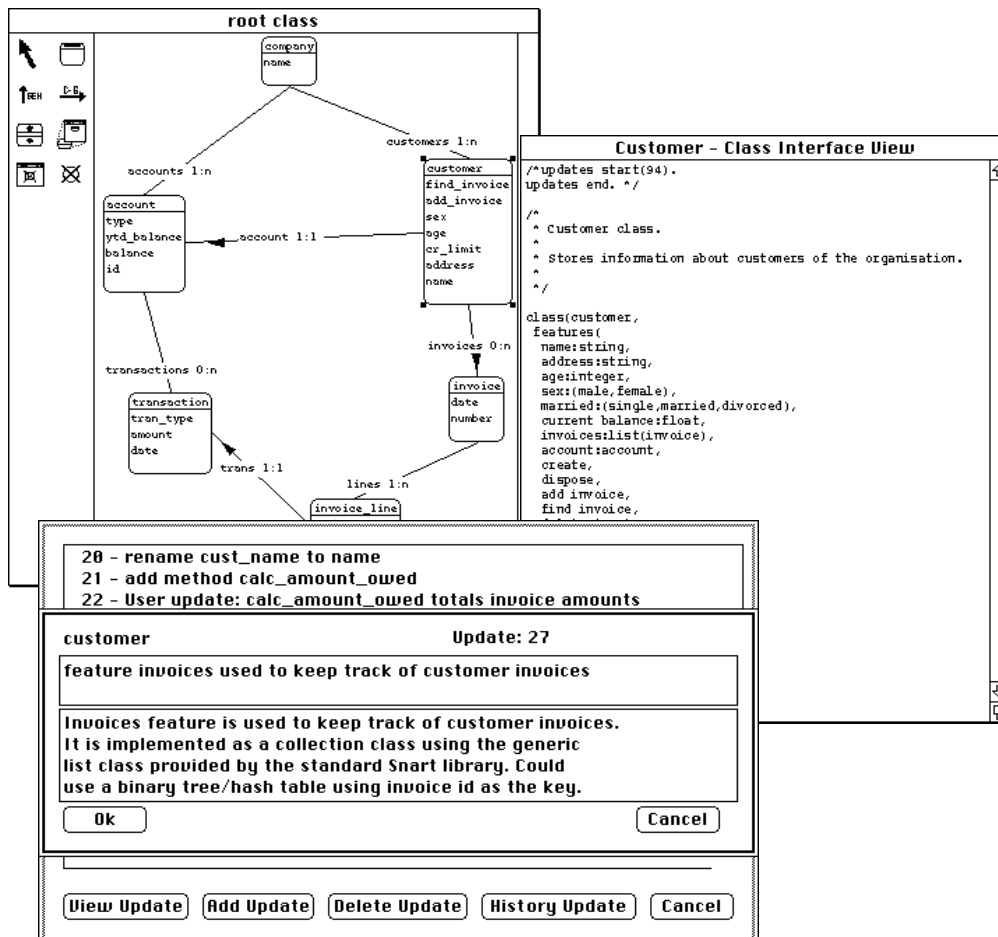


Figure 11. Example of a user-defined change description from SPE.

7. Handling Inconsistencies During Collaborative Software Development

Computer-Supported Cooperative Work (CSCW) systems may use a multi-view editing approach to sharing and modifying information [38, 53, 59]. Inconsistencies between views are often difficult to resolve, as developers may share and modify different versions of the same view in incompatible ways, and may concurrently modify views at different levels of abstractions. We have used the techniques described in Sections 3 to 5 to support a range of inconsistency management techniques for collaborative software development [30].

7.1. Inconsistencies During View Version Merging

CPRG-based environments support the creation and sharing of multiple alternate versions of views i.e. copies of a view which can be asynchronously modified by developers and then merged to produce a consistent, shared software specification. While alternate versions of a view are being modified independently, inconsistencies between these views may easily result. For example, a class method modified by one developer may be deleted or renamed by another. We use the CPRG change description storage mechanism to construct “deltas” between view versions, i.e. sequences of changes made by developers to different alternate versions of a view. Developers may then combine these sequences of view modifications by actioning each change in turn on the original version of a view. The non-sequential undo/redo facility supported by CPRG change descriptions is used to support this. Structural inconsistencies may occur when some of the modifications made by one developer are incompatible with those of another. Similarly, semantic constraint violations may be present in the alternate view versions being merged, or may be present in the new version resulting from the merging process. Both structural and semantic constraints resulting from version merging can be presented to developers using the techniques of Section 5, and developers may resolve these inconsistencies using the techniques of Section 6.

An example from SPE is shown in Figure 12, where two developers have independently modified two alternate versions of an SPE design diagram, and these changes have been merged to produce a new

version of the view which tries to incorporate all of these changes. The developer doing the merging repeatedly selects some or all of the change description presentations for each view, and asks for them to be actioned on the new version being created (using the “Redo” button). In this example, two structural inconsistencies and a semantic inconsistency have been detected during this process, and these inconsistencies have been presented to the developer in the bottom dialogue.

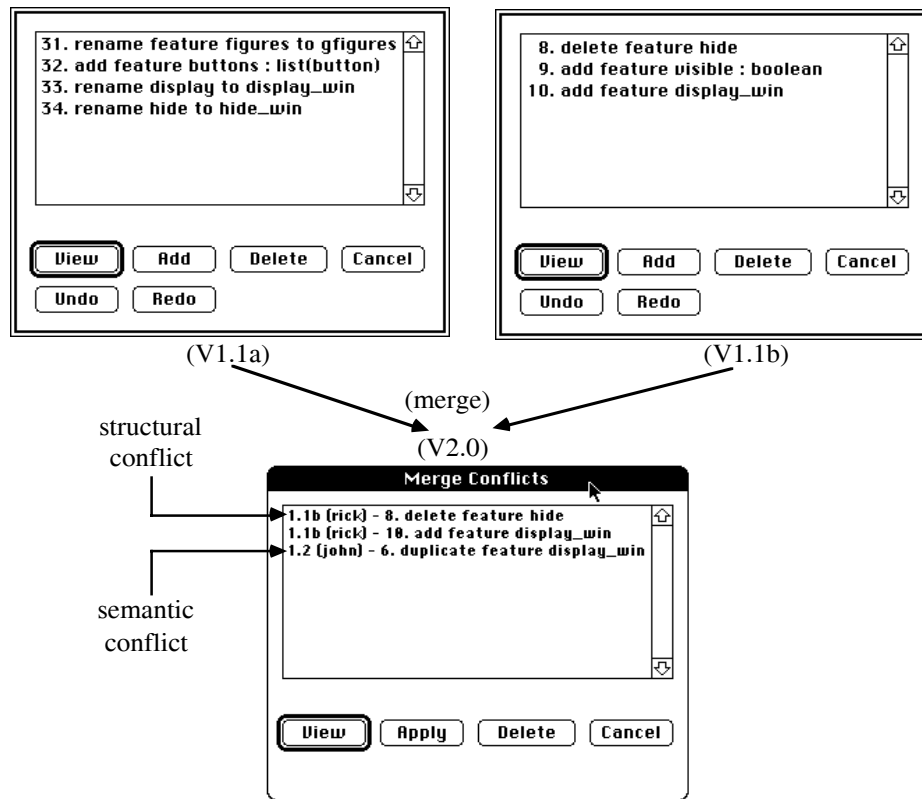


Figure 12. Version merging and presentation of merge conflicts via change descriptions.

7.2. Inconsistencies During Synchronous View Editing

Often developers wish to synchronously or semi-synchronously edit software specification views, particularly those at a high level of abstraction such as analysis and design views. We can use the inconsistency presentation and interaction techniques of Sections 4 and 5 to allow developers to see and act upon the presence of inconsistencies introduced by another developer editing a shared view or a related view. Inconsistencies can be annotated with the name of the developer causing the inconsistency, and presented to users in views, pop-up menus and dialogues as appropriate. Automatically resolved inconsistencies can be stored for perusal by other developers, and automatically modified view items highlighted. We have found assigning colours to each developer to indicate who has last changed what to be useful. Inconsistencies which can not be automatically resolved are presented to developers, and an appropriate resolution strategy negotiated using annotation of change descriptions, textual chats and/or audio conferencing.

Figure 13 shows an example of semi-synchronous editing in SPE utilising inconsistency presentation and interaction techniques. When user “rick” edits his version of the OOA view at the top, changes are propagated to user “john’s” environment (a screen dump from which is in Figure 13). Inconsistencies introduced by Rick’s editing are shown in dialogues and in textual views. John can configure his environment to automatically apply structural inconsistencies to his views. If desired, he could select inconsistency presentations and ask SPE to apply them (if possible), or could negotiate about the inconsistencies presented with Rick. Synchronous collaboration is also supported by SPE; whenever one user changes a shared view, other users synchronously editing see the view updated. Collaborating developers do not get to judge whether or not they want inconsistencies resolved in different ways with synchronous editing, and need to negotiate closely with collaborators to ensure all agree on changes made and any inconsistency resolution techniques used. As SPE associates groups of all inconsistencies and changes made with corresponding views and repository components, developers can peruse these histories

when necessary, and interact with them to reverse or repeat historical modifications and to track long-term inconsistencies.

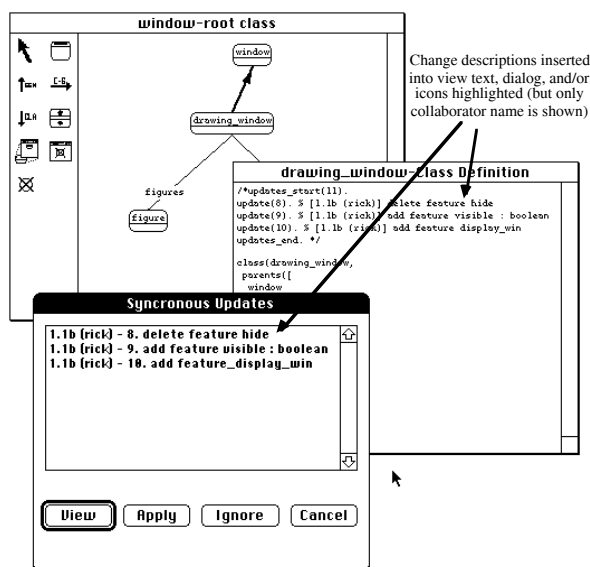


Figure 13. Semisynchronous view editing via change description broadcasting.

7.3. Process-guided Inconsistency Management

Version editing and merging supports quite loose collaborative software development, whereas synchronous and semi-synchronous editing support more tightly-coupled work. We have found that additional process support for group work is also an advantage, enabling inconsistencies generated by multiple developers to be annotated with process stage information and to be grouped by process stage and/or developer causing the inconsistencies. This annotation and grouping of inconsistencies by process stage allows developers to more effectively manage a large number of inconsistencies between multiple views of software specifications. We found that simply grouping inconsistencies by software component or view alone did not scale up for multiple developers working on large projects, as developers often had insufficient information about why inconsistencies had occurred. They also want to view inconsistencies indexed not solely by view or software component, but by the same task in which they are introduced (i.e. those occurring during the same enacted software process model stage).

Our environments support the guidance of software development by the use of Serendipity process models. Serendipity, if in use, is sent change descriptions by environments like SPE, and annotates these change descriptions with information about the enacted software process stage for the developer causing inconsistencies to arise. These inconsistency representations are also grouped with the enacted process stages to give an orthogonal view of work to the standard CPRG component- and view-centred model.

8. Configuring Inconsistency Management Behaviour

While inconsistency management strategies as outlined in Sections 4 to 6 may be hard-coded into CPRG-based environments, they may not always match the tool users' requirements. This is especially true when multiple developers are involved on a complex project and the way different kinds of inconsistencies are managed evolves over time. Environments handling inconsistencies in inappropriate ways may hinder rather than assist software developers. Ideally therefore developers need ways of configuring the inconsistency management techniques they employ. Developers may wish to extend the inconsistency detection schemes used by an environment, indicating additional kinds of structural changes they want propagated between software specification views and specifying additional semantic constraints on software specifications. They may want to change the ways in which inconsistencies that have been detected are presented to them. Similarly, they may wish to have inconsistencies stored in specific places for later perusal, have inconsistencies resulting from other developers' work forwarded to them, or to have their environment carry out specified operations when specific kinds of inconsistencies are detected, to automatically resolve inconsistencies or manage them differently.

For example, when using SPE and Serendipity, structural and semantic inconsistencies are grouped by software component and enacted process model stages. However, developers may wish to group some

inconsistencies in different ways. They may also wish to be informed or have a new inconsistency resolution strategy applied when specific kinds of inconsistencies are detected.

To support developer configuration of inconsistency management in Serendipity, we have developed VEPL, a special-purpose visual event handling language [34]. We have since generalised this language for use in all JComposer-generated systems [33]. The event handling language provides users of CPRG-based environments a flexible mechanism for extending the environment behaviour by: specifying new semantic constraints and structural change propagation mechanisms; detecting certain kinds of inconsistencies which should be acted upon; grouping and presenting inconsistencies in various ways; and automatically performing developer-specified actions when inconsistencies are detected. Inconsistency management strategies can be specified in JComposer using VEPL and thus generated environments have this behaviour hard-coded. Users can also utilise a form of VEPL at run-time, i.e. when using a generated environment, to dynamically reconfigure inconsistency management behaviour.

Figure 14 shows a simple example of a Serendipity visual event-handling model being used to detect and then act upon an inconsistency occurring in SPE. The rectangles are “filters” which take change descriptions and pass on those that match user-specified criteria. The ovals are “actions” which perform simple or complex operations upon receipt of change descriptions (usually from filters). In this example, the development team have determined that multiple inheritance should not be permitted for a particular design. When this VEPL specification detects a developer attempting to use this construct in an SPE view, the change is automatically aborted and the developer informed why this was done. The change is then automatically aborted and the developer informed why this was done. The simple conceptual nature of Serendipity’s filter and action event-handling model was designed to allow end-users of our environments to easily extend their behaviour, rather than having to use complex, low-level textual event handling code.

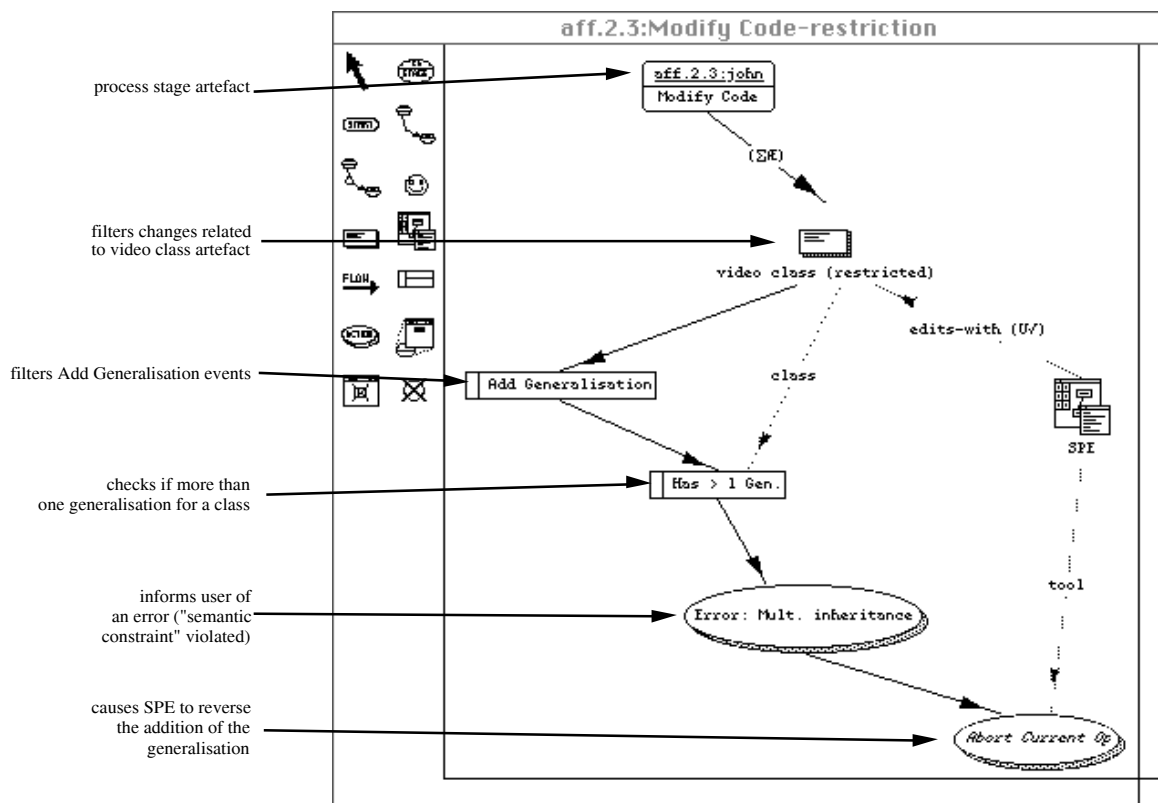


Figure 14. An example of constraining SPE via a Serendipity VEPL view.

The provision of these configuration facilities has interesting implications for collaborative software development. For example, two developers may be collaborating by sharing different versions of a view, but they may specify different filter and actions for the view i.e. different inconsistency management strategies. This may mean changes or inconsistencies made by one developer are not detected and presented to the other developer. There are a variety of ways to address this problem. Developers can be informed when other developers change the inconsistency management behaviour of their environments, as the addition, deletion or modification of filters and actions are themselves documented by change descriptions. Alternatively, additional filters and actions can be specified by project leaders which constrain

the degree of configuration allowed by members of a project team, by constraining the use of filters and actions. In SPE, we document the configuration of inconsistency management by recording filter and action component modifications in Serendipity, and leave it to the project team to negotiate about how inconsistency management configuration should be handled.

An additional benefit of filters and actions is that they support loose collaborative development without necessitating the use of defined software process models. For example, Serendipity can be used with SPE to specify inconsistency management strategies for multiple developers, but without using its software process modelling and enactment facilities. As a codified software process is not used, inconsistencies are not annotated with process stage information nor grouped by process stage. However, filters and actions could be defined to, for example, detect when another developer modifies an abstract specification and to inform other developers of this.

9. Experience and Evaluation

We have deployed the inconsistency management techniques described in the previous sections in the development of a wide range of software development tools. Some examples of these application domains include:

- object-oriented software development [27], including integrated analysis, design, code, Object-Z formal specification, and documentation views.
- collaborative software development tools, including asynchronous and synchronous editors [30] and process modelling and enactment [34]
- integrated EXPRESS-G graphical and EXPRESS textual views [3] for product modelling applications
- integrated OOA/D, EER, and NIAM graphical design tools, with textual relational database views [71], and method engineering facilities via process models and event handling configuration [32]
- integrated graphical and textual specification views for graphical user interface development [29]
- visual programming systems, including visual tool-abstraction [28], user interface [39], and programming systems [48].

While most of these systems have been built using the MViews framework, it takes considerable amount of effort for other developers to build MViews-based tools, due to the complexity of its OO framework. MViews-based tools are also difficult to integrate with third-party systems, and suffer from performance and “scaling-up” problems [30]. We have built JComposer itself, an ER modeller, and a Serendipity-II prototype using JComposer, which has significantly improved the ability of tool developers to quickly design and build tools which use our inconsistency management techniques. JComposer environments are also easier to integrate with third-party tools, as they have a more readily extensible Java Beans-based component implementation.

Our MViews-based environments have been deployed for use on small academic software development projects, and many have been deployed for larger student software development. For example, SPE has been used to facilitate multiple student OO software analysis, design and documentation using its integrated tools. Serendipity has been used by academics and students to describe and enact a wide variety of software and business process models. While MViews-implemented environments have provided inadequate performance and tool integration support for full-scale deployment, we are currently completing the JViews-implemented JComposer and Serendipity-II for use on a “real” multiple person software development project. Usability studies are to be conducted on these tools, in addition to file management, communication and visualisation tools we have been developing, to assess how well they contribute to inconsistency management for multiple person and multiple view software development.

Based on development and use of these environments, we have found that CPRG change descriptions provide an adequate representational mechanism for all inconsistencies we have needed to deal with in these problem domains. The CPRG change propagation and storage mechanisms, and the ability to embody structural and semantic inconsistency detection mechanisms in CPRG components and relationships, have also all proven to be suitable for the inconsistency management needs in our tools.

Users of our multiple-view editing tools like having view inconsistencies representations to see and interact with [27, 29], as these give users immediate feedback on the accuracy and consistency of their views. The presence of inconsistency presentations also assists users in determining where in the view inconsistencies exist and how to begin resolving them. For both system development and maintenance we have found the SPE-style approach of presenting and allowing interaction with inconsistencies in appropriate views to

2.3 Pounamu: a meta-tool for exploratory domain-specific visual language tool development

Zhu, N., Grundy, J.C., Hosking, J.G., Liu, N., Cao, S. and Mehra, A. Pounamu: a meta-tool for exploratory domain-specific visual language tool development, *Journal of Systems and Software*, Elsevier, vol. 80, no. 8, August 2007, Pages 1390-1407.

DOI: [10.1016/j.jss.2006.10.02](https://doi.org/10.1016/j.jss.2006.10.02)

Abstract: Domain-specific visual language tools have become important in many domains of software engineering and end user development. However building such tools is very challenging with a need for multiple views of information and multi-user support, the ability for users to change tool diagram and meta-model specifications while in use, and a need for an open architecture for tool integration. We describe Pounamu, a meta-tool for realising such visual design environments. We describe the motivation for Pounamu, its architecture and implementation and illustrate examples of domain-specific visual language tools that we have developed with Pounamu.

My contribution: Co-developed initial ideas for the approach, co-led design of the approach, co-supervised research assistant, 1 PhD and two Masters students working on software, led evaluation of the platform, wrote substantial parts of the paper, co-lead investigator for funding for the work from Foundation for Research Science and Technology

Pounamu: a meta-tool for exploratory domain-specific visual language tool development

Nianping Zhu, John Grundy, John Hosking, Na Liu, Shuping Cao and Akhil Mehra
Department of Computer Science and Department of Electrical and Computer Engineering
University of Auckland, Private Bag 92019, Auckland, New Zealand
{ nianping, john-g, john, karen }@cs.auckland.ac.nz

Abstract

Domain-specific visual language tools have become important in many domains of software engineering and end user development. However building such tools is very challenging with a need for multiple views of information and multi-user support, the ability for users to change tool diagram and meta-model specifications while in use, and a need for an open architecture for tool integration. We describe Pounamu, a meta-tool for realising such visual design environments. We describe the motivation for Pounamu, its architecture and implementation and illustrate examples of domain-specific visual language tools that we have developed with Pounamu.

Keywords: meta-tools, meta-CASE, domain-specific languages, visual design environments

Introduction

Multi-view, multi-notational Domain-Specific Visual Language (DSVL) environments have become important and popular tools in a wide variety of domains. Examples include software design tools [20], circuit designers, visual programming languages [5], user interface design tools [38], and children's programming environments [7]. Due to the challenge of implementing such tools many frameworks, meta-tool environments and toolkits have been created to help support their development. These include MetaEdit+ [23], Meta-MOOSE [13], Escalante [34], DiaGen [37], GMF [11] and DSL Tools [16], [36].

Current approaches to developing such multiple-view visual language tools suffer from a range of deficiencies. Tools may be easy to learn and use but provide support for only a limited range of target visual environments. Alternatively, the tools may target a wide range of environments but require considerable programming ability to develop even simple environments, providing high barriers to use. Many current frameworks and meta-tools have an edit-compile-run cycle, requiring complex tool regeneration each time a minor notation change is made.

Our experiences in developing domain-specific visual languages for both research and industrial application motivated us to investigate meta-tools able to support exploratory development of such languages while mitigating the deficiencies noted above. Our new meta-tool, Pounamu¹, aims to support users to rapidly design, prototype and evolve tools supporting a very wide range of visual notations and environments. To achieve this we based Pounamu's design on two overarching requirements:

- *Simplicity of use*. It should be very easy to express the design of a visual notation, and to generate an environment for modelling using the notation.
- *Simplicity of extension and modification*. It should be possible to rapidly evolve proof of concept tools by modification of the notation, addition of back end processing, integration with other tools, and behavioural extensions (such as complex constraints).

In this paper we first motivate our work with an example Pounamu-generated visual language application and survey related work. We then overview our Pounamu toolset, describing its visual tool specification

¹ *Pounamu* is the Maori word for greenstone jade, used by Maori to produce tools, such as adzes or knives, and objects of beauty, or *taonga*, such as jewellery.

and modelling support. We discuss Pounamu’s architectural support for tool evolution, multi-user support and a variety of user interface platforms, including web-based diagramming and PDA support. Following this, we describe Pounamu’s design and implementation and illustrate the versatility of the tool using several example applications we have developed. We then evaluate the utility of the tool and conclude with a summary of the contributions of this research and overview of future work directions.

Motivation

Consider the development of an environment to support software process modelling, project management and sketching out of a software design. Examples of diagrams from such a tool are illustrated in Figure 1. Such a tool would provide users with visual tools to model their work process flow (1), describe project scheduling using Gantt charts (2), and software designs using a selection of UML diagrams (3). A team of developers would want multi-user support from such an environment including the ability to version diagrams and at times to collaboratively edit diagrams remotely. For some kinds of diagram it would be useful to provide web-browser or PDA based access e.g. to support on-the-go access to, for example, project management diagrams and reports, as in Figure 1 (2). Models of software designs would need to be able to be exported to other tools e.g. 3rd party UML CASE tools, and code generated from some models. Designing and implementing such an integrated and collaborative toolset is clearly complex; ideally we want this process to be simplified by leveraging meta tool capabilities.

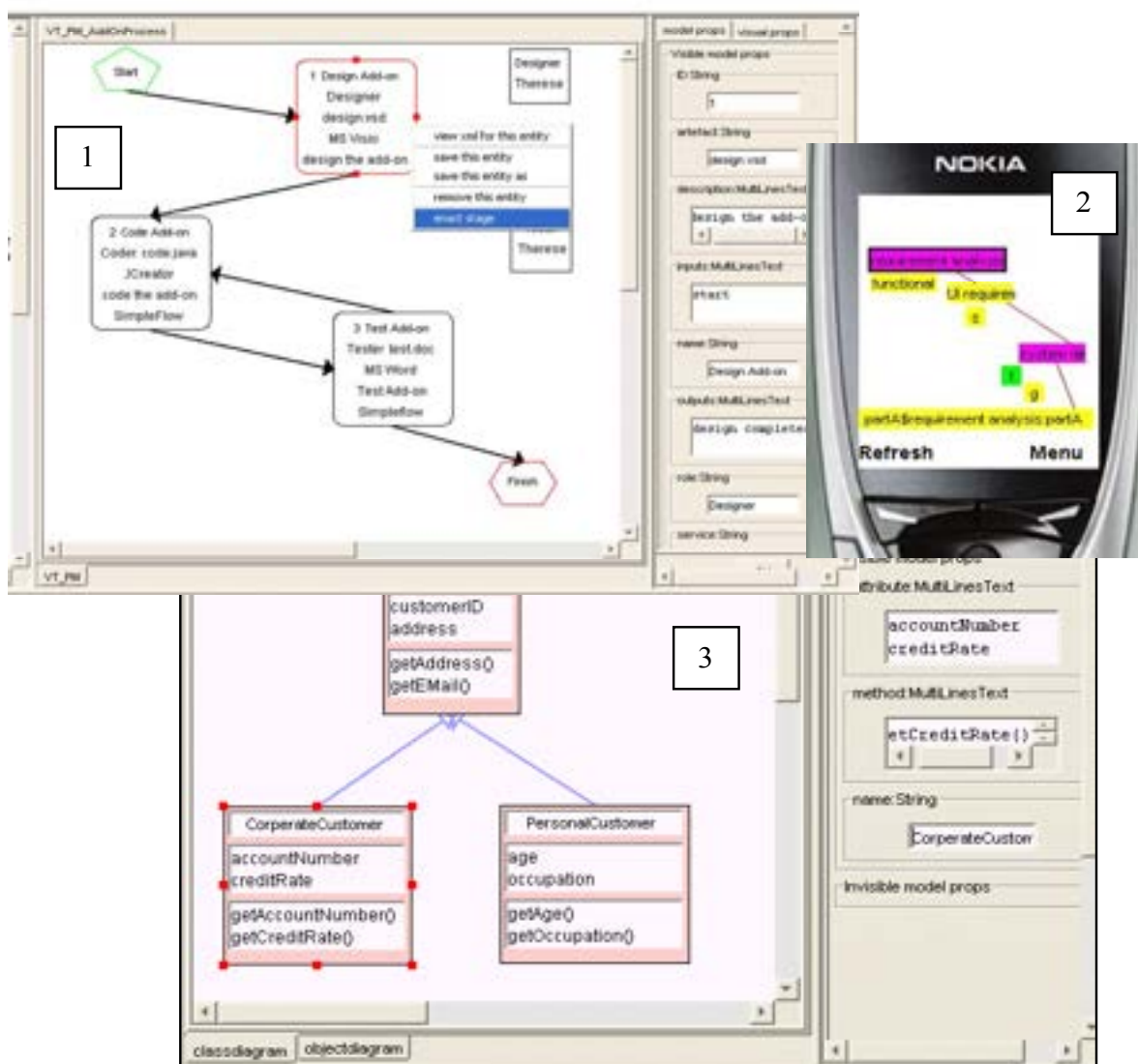


Figure 1. Some examples of DSVL tool diagrams.

While some DSVL tools, such as the ones in Figure 1, may be developed by software engineers, ideally we want to support development and modification of DSVL tools by end-users, so they can make new standalone tools, add new tools to an existing toolset or tailor existing tools to suit their needs. For example, a business process modelling tool might be prototyped by business analysts, a circuit design tool by electrical engineers or a statistical survey design tool by statisticians using their domain knowledge to develop visual notations and tools of relevance to their domain.

Together with the overarching requirements discussed earlier, key requirements for a meta-tool to construct such a range of integrated domain-specific visual language tools include:

- Visual meta-tools to specify DSVL meta-model entities and associations. These abstractions will form the canonical model of information in the target DSVL too.
- Visual meta-tools to specify simple and complex shapes and connectors. These form the building blocks for the diagrams that make up views of the model data structures. When shapes and connectors in a view are modified, the underlying model data structures need to be changed and other dependent views on these structures updated.
- Meta-tools to specify constraint handling for both view editing and model manipulation. Such constraints might include: keeping certain shapes beside or within others as diagrams are edited; computing values based on changes to other values e.g. class inherited properties from other classes in a class diagramming tool; and validation constraints over views and models e.g. all properties of a class must have a unique name.
- Generated tools are able to exchange view and model information structures with other tools e.g. export a model to another design tool, or generate code for a programming environment. The meta-tools should be able to generate tools with an open architecture and should ideally support specification of such import, export and code generation features.
- Generated tools are able to save and load models and views and to share these saved representations among multiple DSVL tools, ideally via a version control server. This supports asynchronous work by multiple DSVL tool users.
- Multiple users are able to collaboratively edit generated DSVL tool views using synchronous groupware support, including group awareness support.
- Generated DSVL tool diagrams are able to be accessed and edited across a variety of platforms, including web browsers, PDAs and mobile phones.

Related Work

Three main approaches exist for the development of the type of visual, multiple view and multi-user environment discussed in the previous section: the use of reusable class frameworks; visual language toolkits; and diagramming or CASE meta-tools.

General purpose graphical frameworks provide low-level yet powerful sets of reusable facilities for building diagramming tools or applications. These include MVC [26], Unidraw [44], COAST [42], HotDoc [4] and GEF [10]. While powerful they typically lack abstractions specific to multi-view, visual language environments, so construction of tools is time-consuming. For example, supporting multiple views of a shared model in GEF requires significant programming effort. Special purpose frameworks for building multiple user, multiple view diagramming tools include Meta-MOOSE [13], JViews [17], and Escalante [34]. These offer more easily reused facilities for visual language-based environments, but still require detailed programming knowledge and a compile/edit/run cycle, limiting their ease of use and flexibility for exploratory development.

Many general-purpose, rapid development user interface toolkits have been developed to reduce the edit/compile/run cycle. Many, including Tcl/Tk [45], Suite [8], and Amulet [38], are suitable for visual language-based tool development. They combine rapid application development tools and programming extensions. However, as they lack high-level abstractions for visual, multi-view environments and tool integration, more targeted toolkits have been produced to make such development easier. These include Vampire [32], DiaGen [37], VisPro [47], JComposer [17], and PROGRES [41]. Some of these use code generation from a specification model, e.g. DiaGen and JComposer. Others, such as PROGRES and VisPro, use formalisms such as graph grammars and graph rewriting for high-level syntactic and semantic

specification of visual language tools. Code generation approaches suffer from similar problems to many toolkits: an edit/compile/run cycle is needed and many tools present difficulties when integrating third party solutions. Formalism-based visual language toolkits may limit the range of visual languages supported and are often difficult to extend in unplanned ways e.g. code generation or collaborative editing.

Meta-tools provide an integrated environment for developing other tools. These include KOGGE [9], MetaEdit+ [23], MOOT [40], GME [27], MetaEnv [1], IPSEN [25], GMF [11], Tiger [12], and MS DSL Tools [16], [36]. Usually they aim for a degree of round-trip engineering of the target tools. Typically they provide good support for their target domain environments, but are often limited in their flexibility and degree of integration with other tools [45]. These integration problems typically occur both at presentation (interface) and data/control levels. Many visual design tools and meta-tools have been enhanced by collaborative work facilities. These range from user interface coupling techniques [8], collaborative editing and authoring [35], and flexible, component-based architectures for collaboration [42][46][35][8]. Most of these approaches provide fixed, closed groupware functionality, however, and we desired more flexible control over target visual design tool collaborative work facilities. In addition, we wanted these capabilities to support integration with existing tools.

A number of researchers have identified the need to support thin-client access to visual design tools and various thin-client software engineering tools have been developed to exploit web-based delivery of information. Some examples include MILOS [31], a web-based process management tool, BSCW [3], a shared workspace system, Web-CASRE [30] which provides software reliability management support, web-based tool integration, and CHIME [21], which provides a hypermedia environment for software engineering. Most of these tools provide conventional, form-based web interfaces and lack web-based diagramming tools. Some recent efforts at building web-based diagramming tools have included Seek [22], a UML sequence diagramming tool, NutCASE [14], a UML class diagramming tool, and Cliki [33], a thin-client meta-diagramming tool. All of these have used custom approaches to realise the thin-client diagramming tool. They also provide limited tool tailorability by end users and limited integration support with other software tools.

In summary, the majority of the approaches we have outlined require detailed programming and class framework knowledge or understanding of complex information models (eg graph grammars). Few of these environment development tools support round trip engineering and live, evolutionary development. Regeneration of code can be a large problem when integrating backend code and most provide very limited, if any, collaborative work facilities. Almost no meta-tools or tool frameworks we are aware of support thin-client diagramming.

Overview of Pounamu

Figure 2 shows the main components of Pounamu. Users initially specify a meta-description of the desired tool via a set of visual specification tools. These define:

- The appearance of visual language notation components, via the “Shape Designer”, which has shape and connector variants;
- Views for graphical display and editing of information, via the “View Designer”;
- The tool’s underlying information model as meta-model types, via the “Meta-model Designer”; and
- Event handlers to define behaviour semantics, via the “Event Handler Designer”, which has visual and textual variants.

Tool projects are used to group individual tool specifications.

The event handler designer allows tool designers to choose predefined event handlers from a library or to write and dynamically add new ones as Java plug-in components. Event handlers can be used to add:

- view editing behaviour e.g. “if shape X is moved, move shape Y the same amount”;
- view and model constraints e.g. “all instances of entity Z must have a unique Name property”;
- user-defined events e.g. “check model is consistent when user clicks button”;
- event-driven extensions e.g. “generate C# code from the design model instance information”; and

- environment extension plug-ins e.g. “initialise the collaboration plug-in to support synchronous editing of a shared Pounamu diagram by multiple users”.

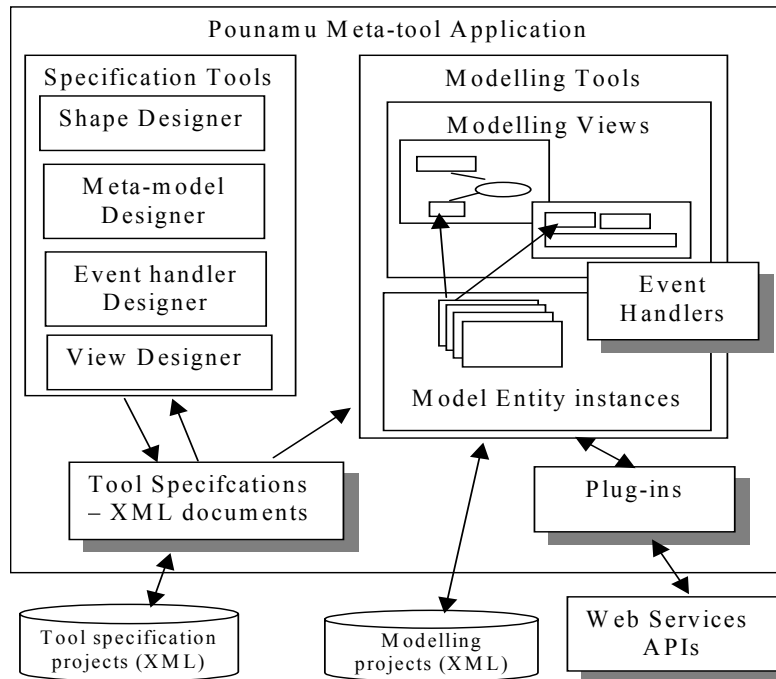


Figure 2. The Pounamu approach.

Having specified a tool or obtained someone else’s tool project specification, users can create multiple project models associated with that tool. Modelling tools allow users to create modelling projects, modelling views and edit view shapes, updating model entities. Pounamu uses an XML representation of all tool specification and model data, which can be stored in files, a database or a remote version control tool. Pounamu provides a full web services-based API which can be used to integrate the tool with other tools, or to remotely drive the tool.

Tool Specification

Figure 3 (a) shows an example of the Pounamu shape designer in use. On the left a hierarchical view provides access to tool specification components and models instantiated for that tool. In the centre are visual editing windows for defining tool specification components and model instances. Here, a shape is defined representing a generic UML class icon. To the right is a property editing panel supplementing the visual editing window. General information is provided in a panel at the bottom.

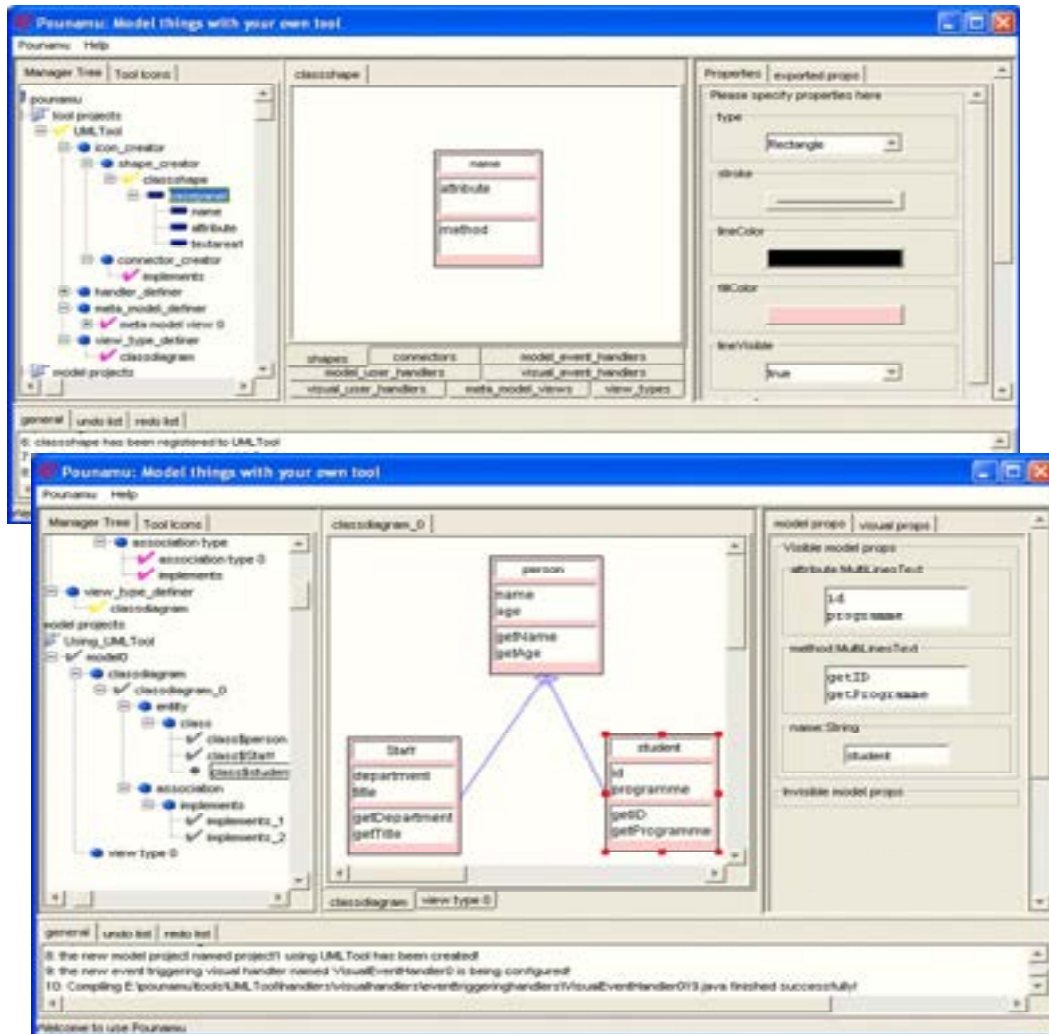


Figure 3. Pounamu in use: (a) specification of a visual notation shape element and (b) modelling using this shape in a UML class diagram tool.

Figure 3 (b) shows a UML class diagramming tool, which uses the shape icon defined in Figure 3 (a) to model a *person* class, and two subclasses *student* and *staff*. The same shape specification could be reused for other modelling tools associated with the same (eg a class element in a package diagram) or different metamodel elements.

The *shape designer* allows visual elements (generalised icons) to be defined. These consist of Java Swing panels, with embedded sub-shapes, such as labels, single or multi-line editable text fields (with formatting), layout managers, geometric shapes, images, borders, etc. For example, the icon in Figure 3 (a) consists of a bordered, filled rectangular panel, with three sub-shapes, a single line textfield for the name, and two multi-line textfields for the attribute and operation parts of the class icon. The property sheet pane (right) allows names and formatting information to be specified for each shape component. Fields that are to be exposed to the underlying information model are also specified using a property sheet tab. Form-based interfaces can also be defined by using a single shape specification defining the whole form.

The *connector designer* allows specification of inter-shape connectors, such as the UML generalisation connector shown in Figure 4. The tool permits specification of line format, end shapes, and labels or edit fields associated with the connector's ends or centre.



Figure 4. Example of the Connector designer.

The underlying tool information model is specified using the *meta model designer*, shown in Figure 5. This uses an Extended Entity Relationship (EER) model as its representational metaphor. This was chosen because the representation is simple and hence accessible to a wide range of users. For example, the meta model in Figure 5 contains two entities representing a UML class and UML object, each with properties for their names attributes and methods, class type etc. An “instanceOf” association links class and object entities and an “implements” association links classes. The meta model tool supports multiple views of the meta model, allowing complex meta models to be presented in manageable segments.

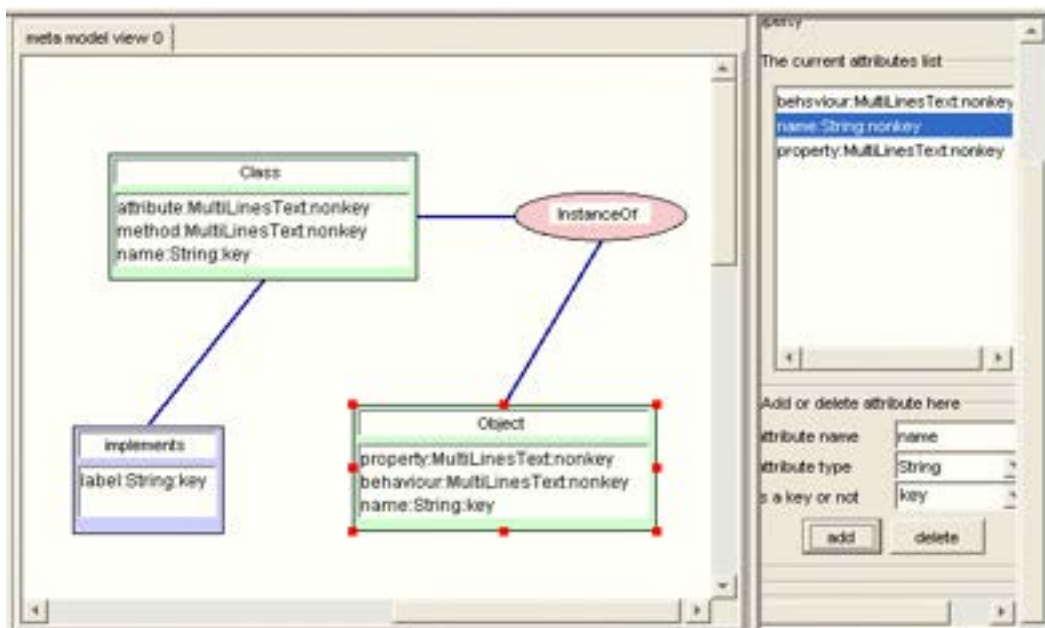


Figure 5. Example of the meta-model designer.

The *view designer*, shown in Figure 6, is used to define a visual editor and its mapping to the underlying information model. Each view type consists of the shape and connector types that are allowed in that view type, together with a mapping from each such element to corresponding metamodel element types. Menus and property sheets for the view editor and view shapes can also be customised using this tool. For example, Figure 6 shows the specification of a simple UML class diagramming tool, consisting of UML class icon shapes, and generalisation connectors. Figure 6 shows that the *classshape* icon maps to the *class* meta-model entity type, and their selected properties map as shown. Mappings supported in this tool are limited to simple 1-1 mappings of elements (single or multi-valued) between view instance and information model instance. More complex mappings can be specified using event handlers as described below.

Multiple view types can be defined, each mapping to a common information model. For example, other view types for sequence diagrams or package diagrams can be defined for the simple UML tool.

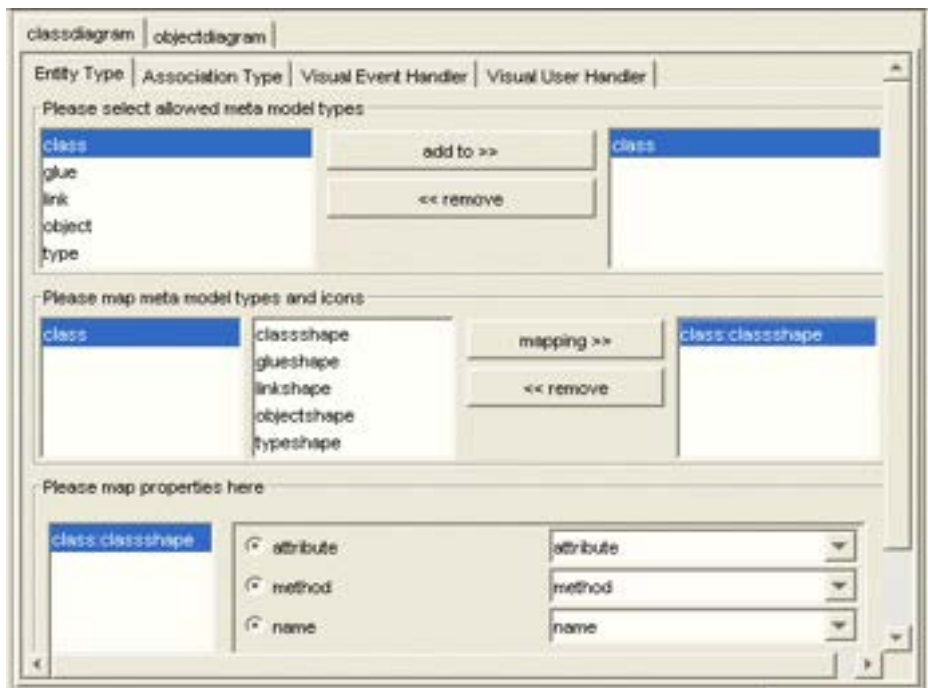


Figure 6. Example of the view designer.

In Pounamu *event handlers* add complex behaviour to a tool using an Event-Condition-Action (ECA) model [28]. Handlers are typically used to add constraints, complex mappings, back end data export or import e.g. code generation, and access to remote services to support tool integration and extension. Each handler specifies:

- the event type(s) that causes it to be triggered, e.g. shape/connector addition/modification, information model element change, or user action;
- any event filtering condition that needs to be fulfilled e.g. property value of shape or entity; and
- the response to that event, i.e. action to take, as a set of state changing operations.

A visual *event handler definer* provides a high-level visual specification based on dataflow for building both simple and complex event handling functionality for Pounamu tools from a set or reusable building blocks. A simple example event handler specification is shown in Figure 7 (a).

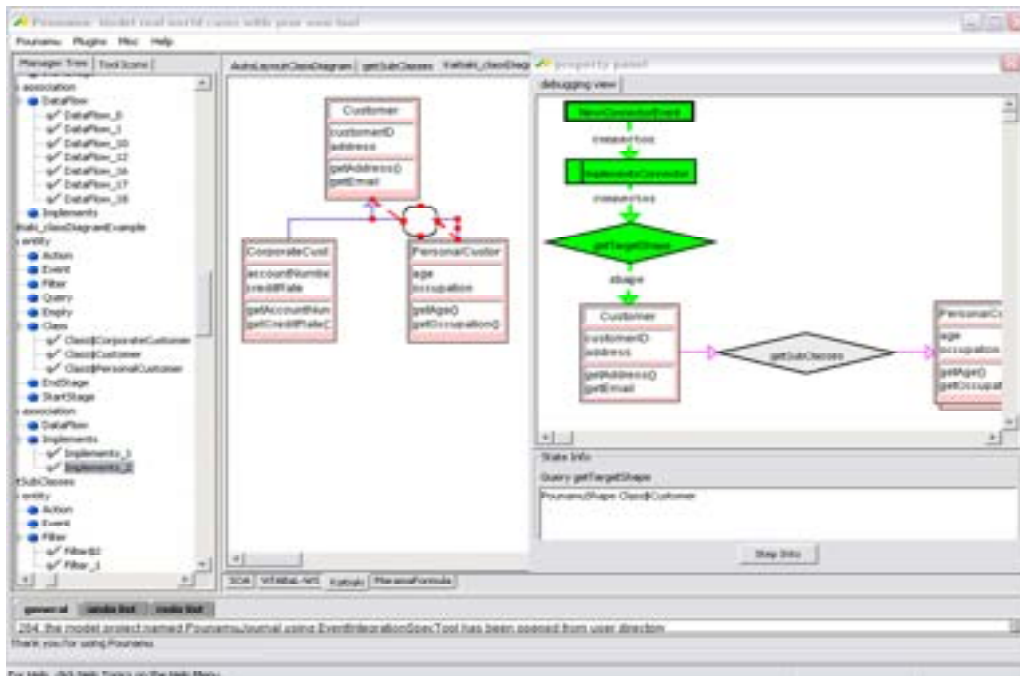
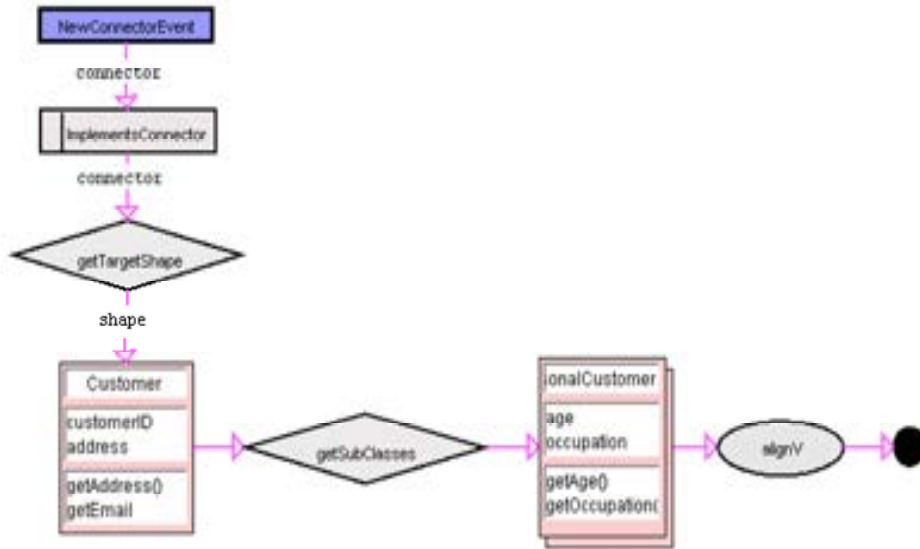


Figure 7. (a) Simple visual event handler specification; and (b) event handler debugging.

In this example an event handler for the UML tool provides automatic layout of subclasses of a parent class for a UML class diagram. This event handler is defined to respond to a built-in Pounamu *NewConnectorEvent*. The modelling constructs contained in this event handler specification include an event, *NewConnectorEvent*, which has the *connector* that has been created as data flowing from it; a filter, *ImplementsConnector* that allows only *Implements* connectors to flow through it; a primitive query *getTargetShape* to locate the end shape of the connector and flow it on to a composed query *getSubClasses* which locates the classes that implement the parent class. The data flows through data propagation links to provide input to connected entities. Type compatibility of the data sender and consumer for each dataflow is statically checked. Also incorporated in the simple event handler example are two end-user target tool icons, one on the data flow between the *getTargetShape* query and the *getSubClasses* query and the other on the dataflow between the *getSubClasses* query and into the *alignV* action. These are annotations that provide a visual indication of the type of data propagating along the links (a class icon, and collection of class icons respectively). We have also developed a visual debug viewer which dynamically annotates an event handler specification view during execution as shown in Figure 7 (b). This includes the visualization of event handler element invocation (by flashing the corresponding graph node) and visualization of data

propagation (by highlighting the dataflow path). The traditional “debug and step into” metaphor is used with step-by-step visualization controlled by menu commands.

For more complex event handlers and to allow expert users to add new condition and action building blocks to Pounamu’s event handler library, event handler code can also be developed using Java. A comprehensive API provides access to the underlying Pounamu modelling tool representation, permitting complex querying and manipulation of tool data. Event handlers may be parameterised and added to one or more tool specification project libraries and then reused by either the visual event handler or code-based event handler specification tools. Code-based handlers are specified using the *handler designer* and included in a tool via the view and meta-model designer tools. A simple example of a code-based event handler being developed is shown in Figure 8. Event handler code is compiled on the fly as the tool is specified or when a tool project is opened.

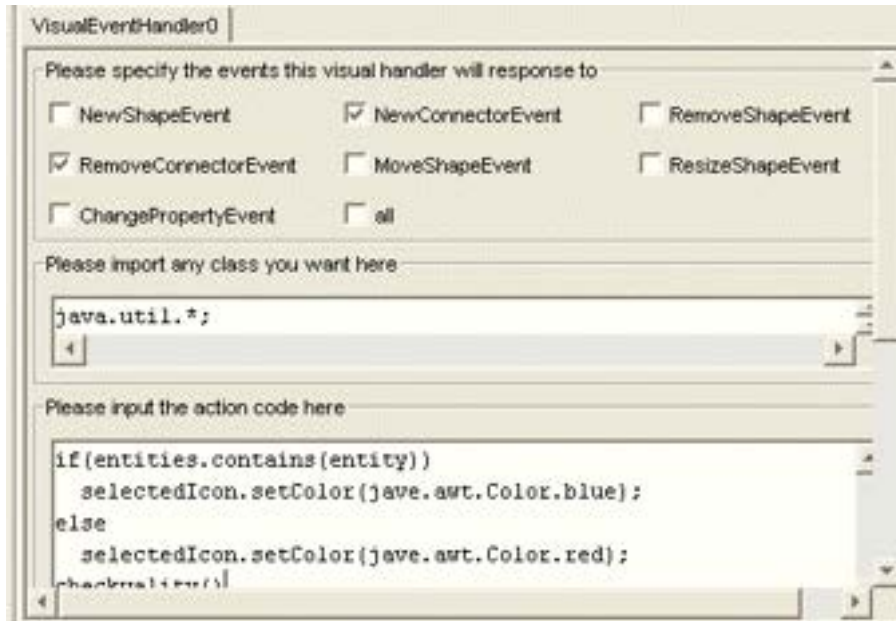


Figure 8. Example of the event handler designer.

Tool Usage

Pounamu automatically and incrementally implements tools as they are specified using the Pounamu metatools. This means tools may be tested and evaluated incrementally as they are being developed, avoiding the compile cycle issues noted earlier and creating a live environment. Generation of the tool happens automatically and immediately following specification of any view editor associated with the tool or when a saved tool project is opened. This provides powerful support for rapid prototyping and evolutionary tool development. Changes to a tool specification may, of course, result in information creation or loss in the open or saved modelling projects e.g. when adding or deleting properties or types. Users can create model views using any of the specified view editors. Reuse is supported by allowing shapes, connectors, meta model elements, and event handlers to be easily imported from other tools or libraries. Multiple tool specification projects may be open when modelling, with specification of parts of the modelling tool coming from different tool specification projects, supporting layered tool development

Each view editor provides an editing environment for diagrams using the shapes and connectors it supports. Consistency between multiple views is implicitly supported via the view mapping process with no programming required to achieve this, unless complex mappings are required that need event handlers to implement them.

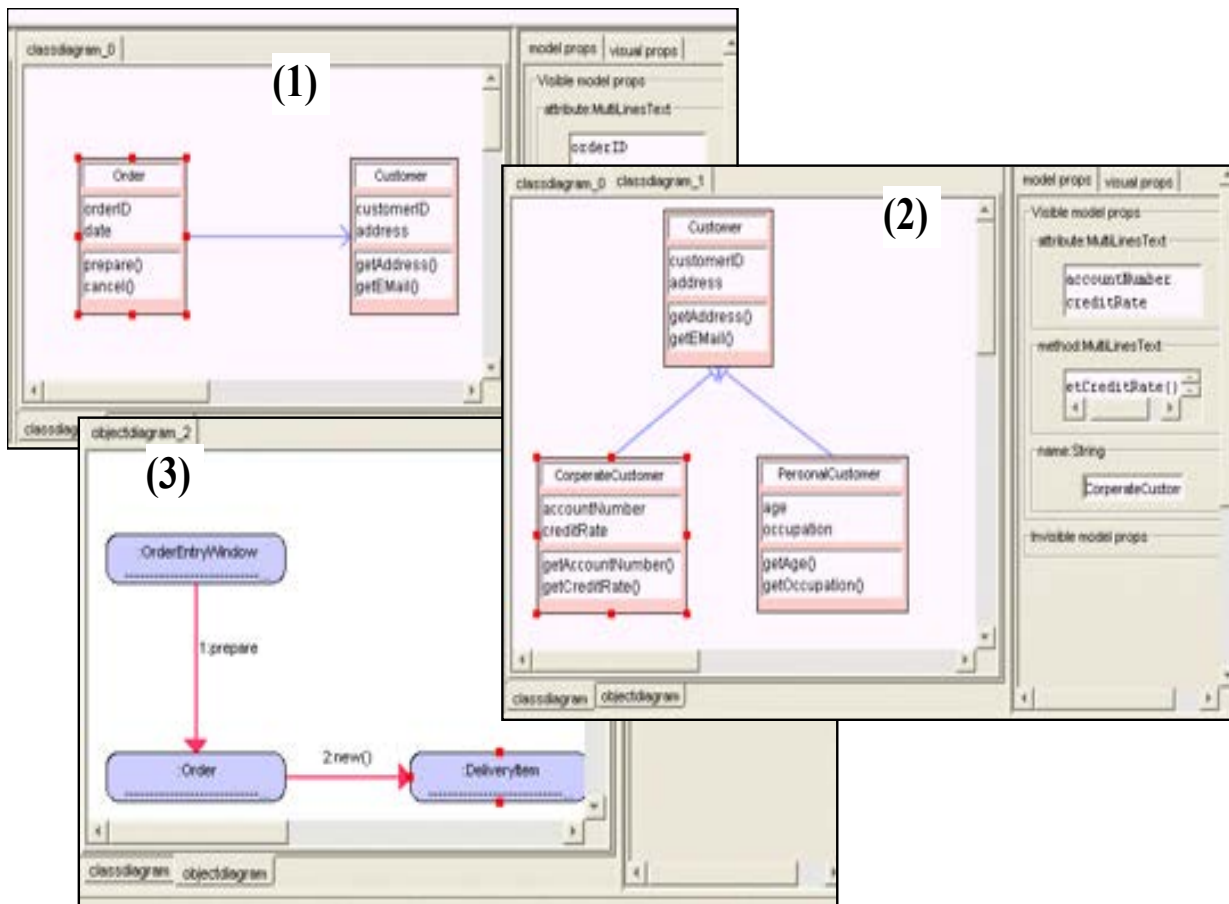


Figure 9. Example modelling tool usage.

Figure 9 shows the simple UML class diagramming tool in use. View (1) shows a simple class diagram. The user has created a class diagram view from the available view types, added two UML class shapes and an association connector, and set various properties for these, including their location and size. View (2) shows another class diagram included in the same project model, reusing the *Customer* class information. Changes to either view, eg addition of a method or change of the class name, are reflected through to the other view. View (3) shows a simplified object diagram view, including an object of class *Order*. Changes to the class name are automatically reflected in this view and only methods defined or inherited by a class may be used in the message calling. The latter is controlled by event handlers managing the more complex consistency requirements.

Having defined a simple tool, and experimented with its notation, additional behaviour can be incrementally added using event handlers to implement more complex constraints. Examples include:

- type checking, e.g. UML associations must be between classes;
- model constraints, e.g. UML class attributes must have unique names for the same class;
- layout constraints and behaviour, e.g. auto-layout of a UML sequence diagram view when edited;
- more complex mappings, e.g. changes to class shape method names automatically modifying method entity properties in the modelling tool information model; or
- back end functionality, e.g. generating C# skeleton code from model instances.

These handlers can be generic for reuse (eg a generic horizontal alignment handler) or specific to the tool. As with other meta specification components, adding or modifying a handler results in “on the fly” compilation of handler code and incorporation of that code into any executing tool instances.

As noted above, back end functionality can be implemented by event handlers. In addition, as all tool and model components are represented in XML format, it is straightforward to implement back end processing

using XSLT or other XML-based transformation tools. This approach can allow back ends to be developed independently of the editing environment enhancing modularisation. An additional approach for implementing back end functionality is via Pounamu’s web services-based API. This exposes Pounamu’s model representation, modelling commands, menu extension capability, etc, permitting tight and dynamic integration of third party tools, and other Pounamu environments. We have, for example, used this API to implement peer to peer based synchronous and asynchronous collaboration support between multiple Pounamu environments, to implement generic GIF and SVG web-based thin client interfaces, to implement interfaces for mobile device deployment, and to integrate a Pounamu based process modelling tool with a process enactment engine [19].

Pounamu’s extendibility also applies to the meta environment itself. For example, we have incorporated support for zoomable user interfaces as an alternative to the conventional editing interfaces described earlier. This extension uses the Jazz ZUI API framework [2] to implement a context and focus metaphor. Figure 10 shows an example of using these zoomable views in Pounamu. The Radar view (left hand side) provides a zoomed-out context for the diagram, allowing the user to zoom the radar view as a whole in and out. A zoomable view (middle, top) allows the user to selectively zoom individual shapes and a Split view similarly provides a selection from the Radar or Zoomable view that can be individually zoomed. Integration of the ZUI framework was simplified by using event handlers to manage and control changes in the ZUI views.

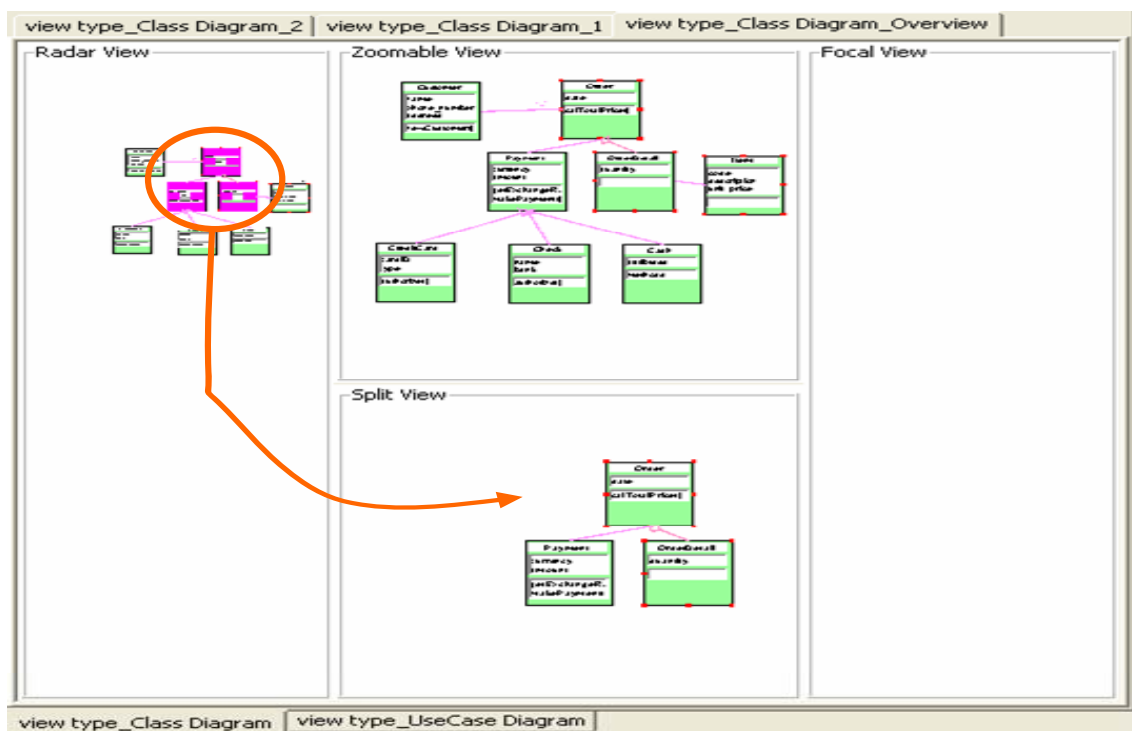


Figure 10. Zoomable views for complex Pounamu diagrams.

We have developed a wide range of exemplar DSVL tools with Pounamu, some of which are illustrated in Figure 11. These include:

- A full UML tool supporting all UML diagram types [48]. This also provides an import/export facility using the XML Model Interchange (XMI) standard allowing models to be imported from and exported from other XMI-compliant UML tools. A code generator takes XMI models from Pounamu and generates Java code that can be further extended by a programming environment.
- A circuit design tool (Figure 11 (a)) providing a CAD-like tool for circuit design.
- A web services composition tool, ViTABaL-WS [29] (Figure 11 (b)). ViTABaL-WS provides web service composition support to business analysts. It provides a tool abstraction composition view and a Business Process Modelling Language composition view both onto a shared model of business processes. The tool generates Business Process Execution Language for Web Services

(BPEL4WS) web service composition scripts which are run via a third-party workflow engine to realise the composed web service specification. The workflow engine uses Pounamu's web service API to support dynamic visualisation of enacted BPEL4WS models as a dynamic debugger.

- A statistical survey design tool SDLTool [24] (Figure 11 (c)). SDLTool provides multiple views describing statistical processes, data and analysis steps. This is used by statisticians to design, enact and process complex statistical surveys.
- A software process modelling and enactment tool IMAL [19] (Figure 11 (d)). IMAL provides multiple users multiple workflow modelling diagrams. These software process models can be enacted by the tool to help support work co-ordination by multiple developers. External tools are invoked via Pounamu's web service support to provide complex rule processing and XML document display and update. Pounamu itself is invoked by a workflow engine via Pounamu's web services API to support dynamic visualisation of enacted work processes.

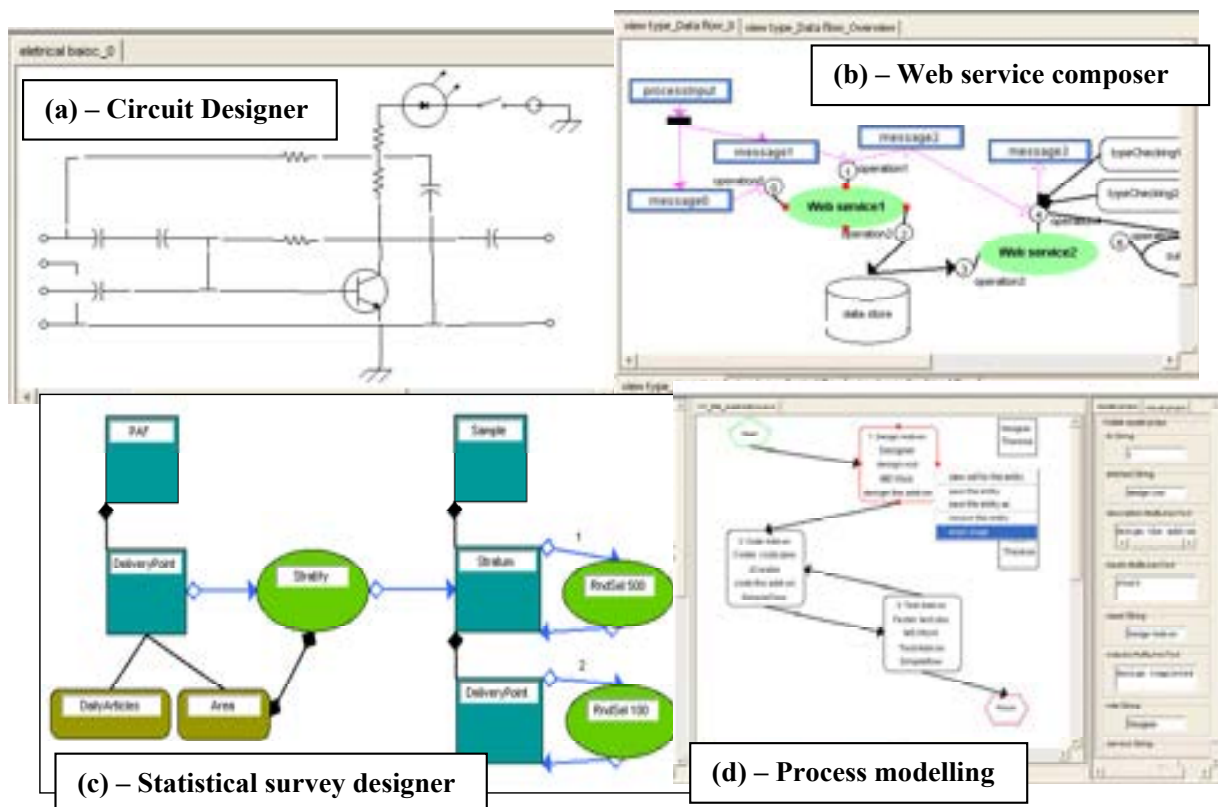


Figure 11. Examples of Pounamu DSVL tools.

We have also used Pounamu as a rapid prototyping tool in a range of industrial applications to assist in the design of visual notations and interfaces for client companies. These applications include: a business form designer; a business enterprise modelling tool; a project management tool with Gantt and work breakdown schedule views; and a web services composition tool. In each case the client companies were able to rapidly explore and evaluate a variety of alternative notational approaches with a low level of investment, hence allowing them to lower the risk of development.

Web, Mobile and Groupware Support

Pounamu-implemented visual design tools may need to be accessed in a variety of deployment scenarios and by multiple users. We have developed a set of plug-in components that use the web services API of Pounamu to extend *any* Pounamu-specified tool with:

- Editable web-based diagram views using either GIF or SVG (Scalable Vector Graphics) images
- Editable mobile PDA/phone diagram views using Nokia's MUPE framework
- Collaborative editing of diagrams

- Asynchronous version control and version merging support for diagrams, along with CVS repository management of versions

An example of the web-based, thin-client editing interface for Pounamu tools being used is shown in Figure 12 (left). This allows a group of users to interact with Pounamu views via web browsers and standard web software infrastructure. SVG image diagrams support browser-side drag and drop of diagram component using scripting.

A single Pounamu instance runs as an application server, while a set of Java servlets provide the web component implementation technology. No code changes are necessary to support this web-based diagramming, as the plugin is generic for any Pounamu specified tool; the servlets use the web services API of Pounamu to obtain available Pounamu tools and convert the view elements into GIF images or SVG encodings automatically.

We have developed an alternative set of software components using Nokia's MUPE framework that allow users to view and edit diagrams on a wireless PDA or mobile phone. This uses a similar approach, where the MUPE server components communicate with Pounamu via its web services API and generate MUPE XML mark-up for the view user interfaces. Figure 12 (right) shows an example of a project management Gantt chart diagram being browsed and manipulated on a mobile phone. The MUPE servers implement a flexible zooming mechanism which magnifies selected items dynamically to mitigate the small screen size of the mobile devices. Again, the plugin is generic, meaning that any Pounamu tool can be deployed using this technology without additional programming.

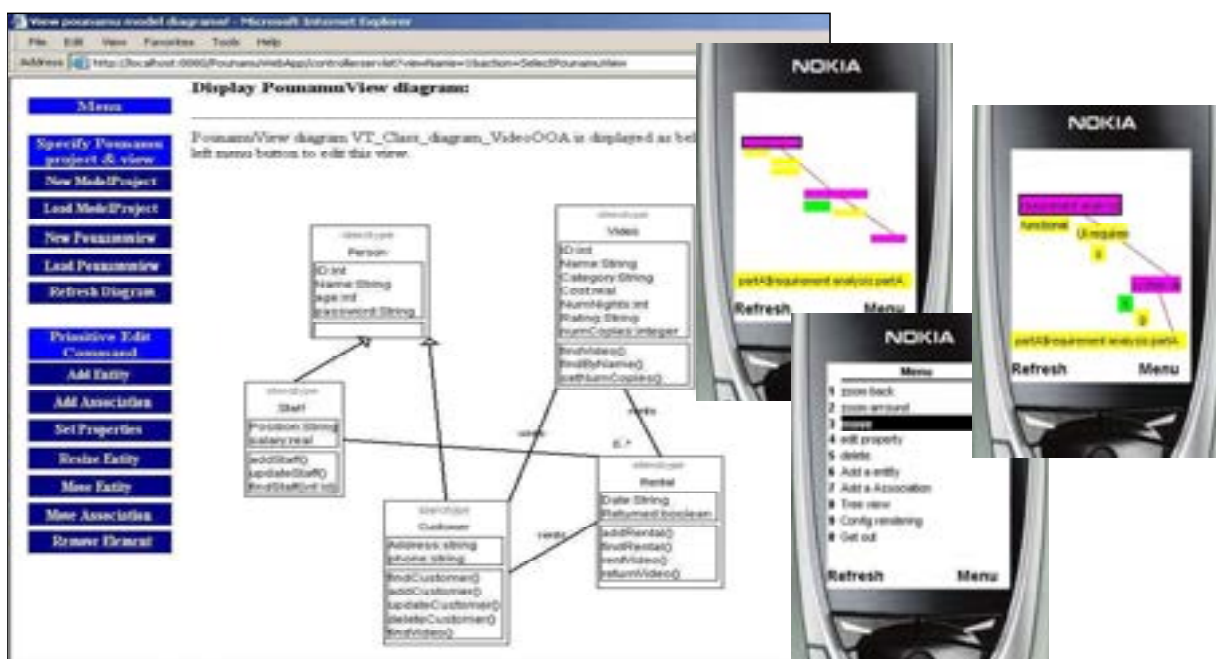


Figure 12. Thin client, web-based editing interface generated for Pounamu UML tool.

Collaborative work is supported in three ways: using one of the thin-client (web or mobile) user interfaces; using a set of plug-in synchronous editing components; or using a set of asynchronous version control and merging components. These all make use of Pounamu's web services API to support sending and receiving of collaborative editing messages between multiple Pounamu instances, synchronising views. They also support check in and check out of views from a CVS repository and visual differencing and merging support. An example of collaborative editing is shown in Figure 13 (a), where two users are editing a diagram together. Changes made by one user are sent to the other user's Pounamu and automatically applied. Changed diagram content is highlighted. A visual differencing algorithm uses a similar approach to support asynchronous version comparison and merging. An example of differencing two UML diagrams is shown in Figure 13 (b) where two versions have been compared and differences between them highlighted in the diagram.

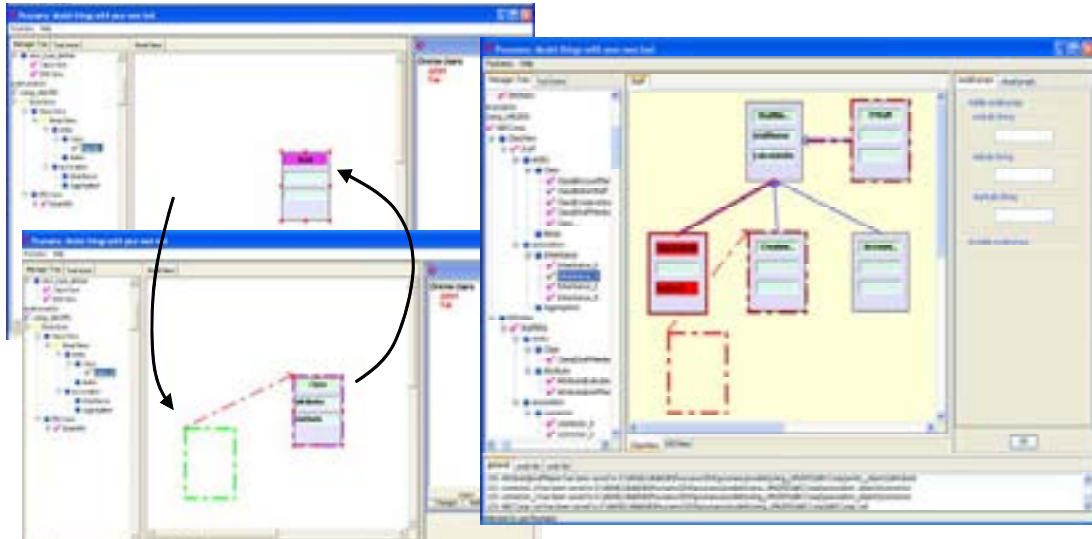


Figure 13. (a) Collaborative editing and group awareness; (b) version differencing.

Implementation

Pounamu is implemented in Java, using the Xerces XML parsing libraries, Swing user interface packages, and Java web services development toolkit. The main components of Pounamu are outlined in Figure 14. The Pounamu design and modelling tools use Java Swing to implement their user interfaces. The design tools create tool specifications, a set of shape, connector, view, entity, association, project and event handler types, which when composed together form a Pounamu tool specification. The modelling tool uses a model-view-controller architecture to provide data representation of entity, association, shape, connector and view instances (model); Swing-based representations of these model instances (view); and Command objects that modify the model state, created by interaction with the visual components (controller).

The Java JAX XML API and Xerces Document Object Model (DOM) framework were used for representing both tool specification data and modelling project data as in-memory XML data structures. The Java file management APIs were used for information storage and retrieval of tool specifications and modelling tool data to/from the XML format. A specialised class loader is used to support on-the-fly event handler class compilation and reloading, enabling dynamic code addition and replacement in the tool.

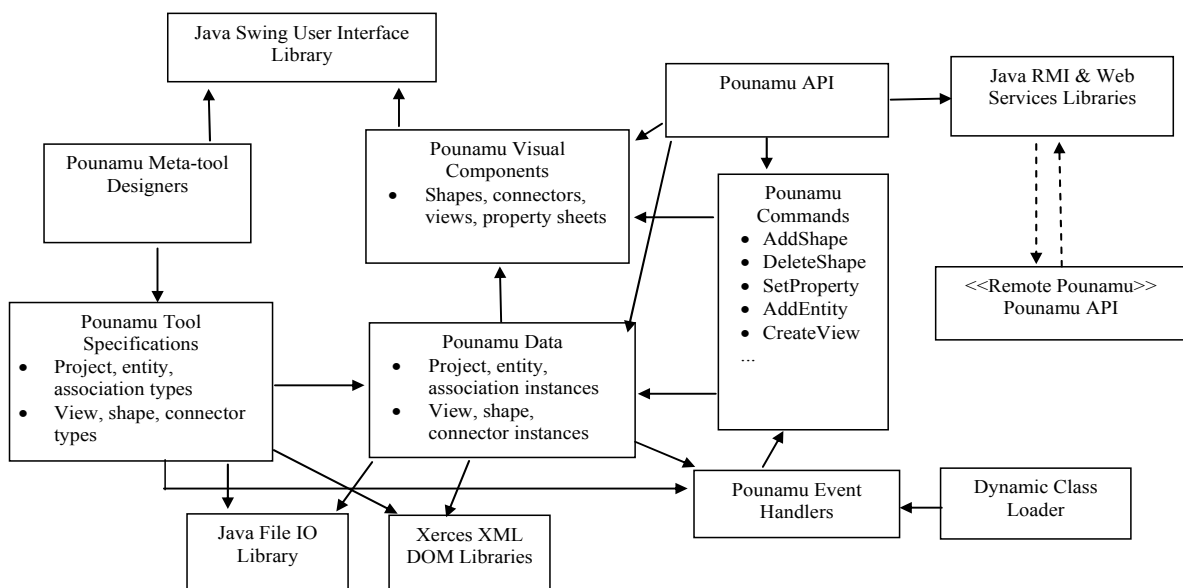


Figure 14. The component structure of Pounamu.

Specification and modelling tool information is represented in an XML format, both internally using a DOM API and externally in either XML files or a database. We chose an XML based representation to permit ready extension to the tool and model formats, ease of exchange with other tools, and the ability to use existing translation support tools. These XML formats also allowed us to adopt a web services-based API extension and integration approach for Pounamu. We have used XSLT translation scripts to support translation of Pounamu model and view data into other formats for information import and export [43].

The Java web services development toolkit was used to implement a web services API for Pounamu. This API provides external tools with the ability to query Pounamu model and diagram information in an XML format and to create and run Pounamu editing commands to update these data structures remotely. This web services API has been used to build the generic collaborative editing component for the Pounamu modelling tool which transmits XML-encoded diagram update messages between different instances of Pounamu. It has also been used to build the set of Java servlets used to provide the generic Pounamu thin-client interface plugin and a similar set of servlets for the MUPE-based mobile interface.

Evaluation

Evaluating a meta tool such as Pounamu is not a straightforward task due to the multiple points of view involved (tool developer, end user of developed tool, usability, utility, etc). Our approach has been to evaluate Pounamu at several levels and through a variety of mechanisms. These include:

1. Two large group experiments, spaced nearly two years apart, where participants (approximately 45 in each case) constructed a domain specific visual language tool and then surveyed. Our aim in each case was to use the feedback to improve the tool, and significant enhancement was undertaken between the two experiments.
2. Qualitative feedback, in the form of experience reports, from a smaller number of developers who used Pounamu to develop more substantial applications, such as the ones in Figure 11. These were used to assess whether perceptions altered as more substantial applications (with, for example, more complex back end integration requirements) were developed.
3. Small end user and cognitive dimensions [15] evaluations of the various Pounamu extensions, such as thin client and collaborative work support. These were primarily used to compare utility and usability of these extensions against the core Pounamu toolset. Many of these evaluations have been reported in detail elsewhere.
4. Small end user and cognitive dimensions evaluations of substantial applications developed using Pounamu. These were used to evaluate whether end users found Pounamu generated tools to have good usability characteristics. Many of these have also been reported in detail elsewhere.

Large group experiments

In each experiment around 45 participants, who were graduate-level students, were asked to construct a DSVL tool of their own choosing, but with at least a minimal set of required components, such as numbers of icons, views, handlers, etc so that tools with a realistic level of complexity were designed and constructed. Participants were given two weeks elapsed time (i.e. alongside other obligations) to complete application development; they were then surveyed using a set of open ended questions to qualitatively elicit strengths and weaknesses of Pounamu to construct the desired DSVL tool. The surveys, one undertaken in August 2004 (46 participants) and the second in May 2006 (45 participants), emphasised elicitation of weaknesses as their primary intention was to provide feedback to be used in improving Pounamu, hence the responses observed tended to describe generic strengths of Pounamu, with more detail on specific weaknesses.

Generic strengths emphasised by respondents in both surveys included: the rapidity with which tools were able to be constructed; the extensibility and customisability of the generated tools; the low learning curve needed to use Pounamu effectively; and the usefulness of being able to update tool definitions on the fly as iterative development was undertaken.

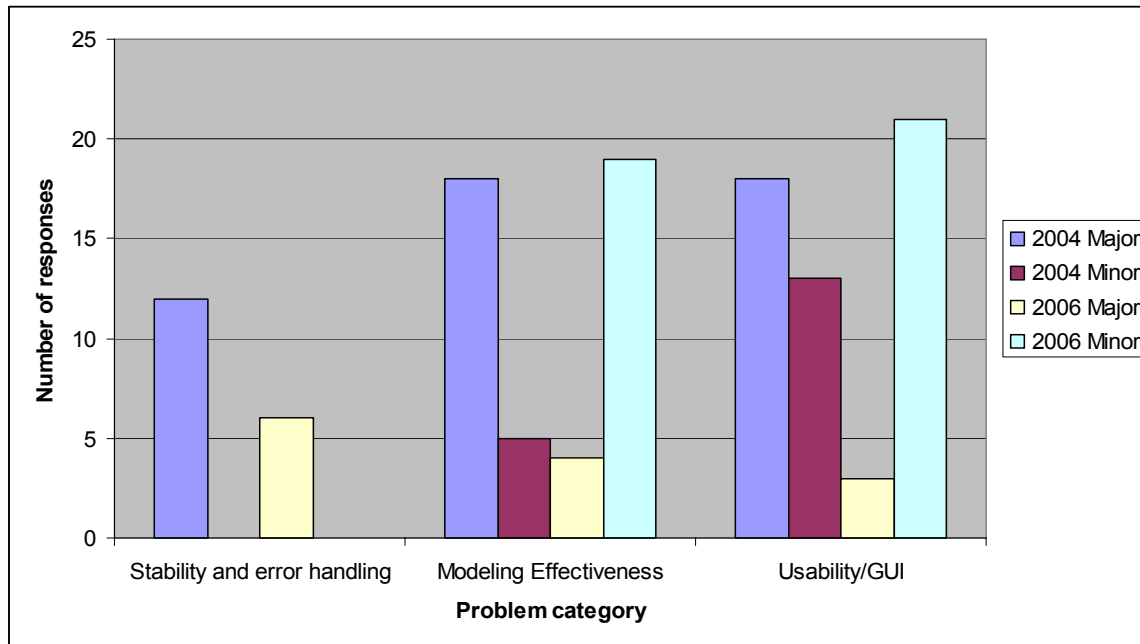


Figure 15: Problems identified in 2004 and 2006 surveys

Figure 15 charts the number of responses concerning identified weaknesses in each survey aggregated into three categories and subcategorised as “major”, i.e. a significant weakness, or “minor”, i.e. an issue causing irritation but not significantly affecting functionality. General weaknesses identified in the first experiment focused on issues of: stability of the software; inconsistency of the interface compared to other tools; lack of documentation, particularly for the API; difficulty of event handler specification; and weak error handling, all of which might be expected of the, at that stage, proof of concept prototype. Issues of stability, documentation, event handler specification and error handling were largely addressed in changes made to the system before the second experiment, which showed much less concern by participants on these issues. For each category the number of major problems identified was significantly lower in the second survey. Many minor usability issues were also identified by participants, such as the size of text fields for entering Java code for event handlers, some clumsiness around specification of icons, and the response time for some elements of functionality. The second experiment showed increased concern by participants in such issues, primarily, we speculate, because a lack of concern over major issues permitted them to focus more readily on more minor limitations of the meta-tool. Most issues identified concerned usability of the tool specification components, with very few issues concerns with the usability or efficacy of the generated tools themselves, nor of the representational power of the meta model.

A key feature for us is the efficacy of our evaluation-cycle based quality improvement approach which has demonstrated significantly enhanced user perception of Pounamu following attention to issues raised in the first large-scale user evaluation.

Large application developers

A smaller group of 8 developers have used Pounamu to develop more substantial applications, typically as an element of a larger research project, and over an extended period (several months at least). In each case, the developers provided detailed comments on the efficacy of Pounamu as part of a wider ranging implementation report. These qualitative comments were summarised and categorised in a similar manner to the large experiments. The small number of participants makes for less depth in the results, however, they are sufficient to assess any significant change in perception with increased application size. The general strengths identified were the same as for the large group experiments, but with more emphasis on the speed of development and the extendibility and customisability of the generated tools. Far fewer weaknesses were identified; familiarity with Pounamu appears to have mitigated many of minor difficulties identified in the large group experiments. Some issues around stability and performance were identified by those using Pounamu in its early stage of development, in common with the first group experiment, but those using later versions did not report the same issues. In summary, our results suggest that developers

using Pounamu to construct larger applications are somewhat more favourably inclined than those developing small applications as in the former case the benefits of efficiency of construction significantly outweigh minor usability issues.

Usability of Pounamu extensions

We conducted user surveys of the thin-client diagramming plug-ins (9 participants) and the collaborative editing plug-ins (10 participants) for Pounamu both using a UML modelling tool developed using Pounamu as a vehicle for the experiments. These are reported in detail in [6], [35]. Our primary aim was to assess efficacy of the extensions over that of the standard Pounamu thick client. Users performed similar design and revision tasks by themselves and in small (2-4 developers) groups. Feedback was generally very favourable, with users appreciating the ease of update of tools via the use of a shared web server and accessibility of diagrams via web browsers. The collaborative synchronous editing capabilities of Pounamu were found to be sufficient for basic revision and diagram construction tasks with the group awareness capabilities particularly praised by end users. The asynchronous versioning and merging support also received good feedback from users [35]. In addition to the usability evaluations, we performed a cognitive dimensions analysis of the versioning and merging plug-ins as well as assessing them against Gutwin's groupware framework [18].

Usability of substantial tools constructed using Pounamu

For many of the exemplar systems described earlier (and others) we have carried out a combination of survey-based end user evaluations of the application visual language environments and cognitive dimensions-based evaluations of the visual environment interaction and information presentation features. These evaluations have each been reported in detail elsewhere and include (in order of application development): the IMÅL distributed process modeling and enactment tool (reported in [19], 8 survey participants); a prototype UML tool (reported as a component of [35], 10 survey participants); and the SDL statistical survey specification tool (reported in [24], 8 survey participants). We summarize these evaluations here as evidence that tools implemented using Pounamu provide good usability characteristics.

We evaluated both IMÅL's modelling capabilities and its presentation of enacted process stage information in Pounamu modelling views [19]. Users reported they found the visual modelling facilities to be good, the multiple view support helpful, and consistency management between different views and error reporting to be good. However they found some modelling functions, such as resizing, to be ungainly, a lack of drag-and-drop creation of model elements, and some bugs in the modelling environment, all problems we have since addressed (IMÅL was developed using the same version of Pounamu as was used in the first group experiment). Our cognitive dimensions analysis of the process modelling tool found a good closeness of mapping of the visual notation to the problem domain, high consistency of views and few hard mental operations were needed to build models. However, at times there is insufficient visibility and juxtaposability. For example two views could not be seen at the same time. These findings have also led to a number of further enhancements to Pounamu, particularly in its view management facilities.

Our user survey of the prototype UML software design tool included experienced, industry modellers as well as novice users. We had users perform selected modelling tasks including creating new designs and revising designs. We also had users make small modifications to the UML design tool notation and meta-model using Pounamu's tool designers. Our Pounamu-built UML design tools were found to be both usable and appropriate to their task. Some problems were encountered with non-standard notational symbols and limited editing capability in some diagram types, especially UML sequence diagrams. These limitations arose from limitations in Pounamu's shape specification and editing event handler control. We have since enhanced both facilities and substantially improved the usability of the UML tools. Users found modifying the notational symbols used very straightforward with Pounamu's shape and connector designers. However they found modifying the meta-model and event handlers to be more difficult and desired improved support for these activities.

The SDL tool used the most recent version of Pounamu in its development and hence is most representative of the current tools state. Participants in the end user study were 8 senior undergraduate statistics students, who were given a tutorial introduction to SDL, prior to completing a series of tasks, followed by a survey including both closed and open-ended questions. Questions examined tool usability (90% positive) and

notation usability (75% positive), while observations of end user performance assessed diagram comprehension (75% good or better) and task completion (only 12% incomplete). Open ended responses focussed almost entirely on the statistical survey modelling ability of the tool rather than any usability or efficacy issues that could be related back to use of Pounamu. This was particularly pleasing as the discourse with participants ended up being entirely focussed on “real” end user requirements rather than technology difficulties imposed by the use of the metatool.

Assessing Pounamu against our two requirements, simplicity of use and simplicity of extension, we can conclude that Pounamu permits rapid development of a DSVL environment for a simple version of the supported notation, satisfying our first requirement. Many of the exemplar tools implemented with Pounamu have then been iteratively expanded in a manner matching the second of our requirements. For example:

- elaboration of notations, such as expansion of the range of UML diagrams supported in the UML tool has been carried out even while the tool has been in use, permitting exploratory DSVL development
- addition of event handlers for complex constraint management, particularly for visual constraints and for consistency management between elements in the information model. For example, our Traits modelling tool used this for generating a combined conflict free method list
- integration of backend code generation support has been provided for the UML, web services and process modelling tools. These have adopted approaches of transforming the saved Pounamu XML model structures into respectively Java code, BPEL4WS scripts, and XML data for display by Microsoft InfoPath.
- use of the web services API to integrate the process modelling tool with a distributed process enactment engine and Microsoft InfoPath.

The extended entity relationship based representation chosen for the tool information model, while simple, has so far proved adequate for most tools. In particular it was able to directly support implementation of the substantial UML meta model, itself a substantial subset of the OMG UML meta model [39], for the UML tool. The simple mapping representation supported by the view specification tool was adequate for most basic view-to-model data mappings with so far only a few, such as more complex UML and process flow diagrams, requiring more complex mappings implemented using complex event handlers. The visual event handler definer has reduced the need for end users to use the complex API-dependent Java coding of constraints for tools.

Summary

We have described Pounamu, a meta tool for specifying and implementing multiple-view multiple-notation diagramming tools. Our primary aim in developing Pounamu was to provide both ease of use and simple extendibility of both the meta tool and generated tools. These have been achieved through a simple conceptual base and a simple set of specification tools, together with live, incremental implementation, and a heavy emphasis on backend integration capability. The latter has permitted ready implementation of significant and generic extensions such as thin and mobile client deployment, synchronous and asynchronous collaboration support, and zoomable user interfaces, together with tool specific code generation and 3rd party tool integration implementations. We have used Pounamu to develop a broad range of DSVL applications both for academic/research use and industrial purposes and over a large end user base. Informal feedback and formal evaluation of the tools has been very positive.

In current work, we are developing a formula definer to augment Pounamu’s meta-model and view designers, allowing tool developers to specify formulae over both model and view data structures combined with a one-way constraint system to compute values during tool usage. This will allow simpler specification of computed values and some forms of constraint handling within Pounamu tools. In addition we are developing a more complex view specification tool, allowing many-to-many mappings between view shapes and connectors and model entities and associations to be specified. This will again make it easier for tool developers to build more complex view-model mappings without resorting to using complex event-driven handlers. A set of plug-ins for the Eclipse open-source IDE to allow Pounamu tool specifications to be used to generate Eclipse graphical editors is under development, as are further thin-

client visualisation tools using VRML to produce 3D representations of complex Pounamu views, supporting complex design visualisation tasks.

References

- [1] Baresi, L., Orso, A. and Pezze, M. Introducing Formal Methods in Industrial Practice. In *Proc. ICSE 1997*, Boston MA, 1997, ACM Press, pages 56–66.
- [2] Bederson, B., Meyer, J. and L. Good. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java, in *Proceedings of 2000 ACM Conference on User Interface and Software Technology*, ACM Press, 171-180.
- [3] Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. (1995): Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. *Proc. of the 4th International WWW Conference*, Boston, MA, December 1995.
- [4] Buchner, J., Fehnl, T., and Kuntsmann, T., HotDoc a flexible framework for spatial composition, In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, IEEE CS Press, pp. 92-99.
- [5] Burnett M, Goldberg A, Lewis T (eds) *Visual Object-Oriented Programming*, Manning Publications, Greenwich, CT, USA, 1995.
- [6] Cao, Shuping, Thin-client interface design for the Pounamu Meta-Case Tool, MSc Thesis, Department of Computer Science, University of Auckland, 2004, 164pp.
- [7] Cypher, A. and Smith, D.C., KidSim: End User Programming of Simulations, In *Proceedings of CHI'95*, Denver, May 1995, ACM, pp. 27-34.
- [8] Dewan, P. and Choudhary, R. 1991. Flexible user interface coupling in collaborative systems, *Proceedings of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.
- [9] Ebert, J., Süttenbach, R., and Uhe, I., Meta-CASE in practice: a case for KOGGE, In *Proc. CAiSE'97, LNCS 1250*, 203-216.
- [10] Eclipse, Graphical Editing Framework, www.eclipse.org/gef, accessed 2 October 2006
- [11] Eclipse Graphical Modelling Framework, <http://www.eclipse.org/gmf/>, accessed 13 October 2006
- [12] Ehrig, K., Ermel, C. Hänsen, S. and Taentzer, G. Generation of Visual Editors as Eclipse Plug-Ins, *Proc. 2005 ACM/IEEE Automated Software Engineering*.
- [13] Ferguson R, Parrington N, Dunne P, Archibald J, Thompson J, MetaMOOSE-an object-oriented framework for the construction of CASE tools: Proc Int Symp on Constructing Soft. Eng. Tools (CoSET'99) LA, May 1999.
- [14] Gordon, D., Biddle, R., Noble, J. and Tempero, E. (2003): A technology for lightweight web-based visual applications, *Proc. of the 2003 IEEE Conference on Human-Centric Computing*, Auckland, New Zealand, 28-31 October 2003, IEEE CS Press.
- [15] Green, T.R.G and Petre, M, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *JVLC 1996 (7)*, pp.131-174.
- [16] Greenfield, J., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, <http://msdn.microsoft.com/vstudio/DSLTools/> 2004.
- [17] Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *J. Information and Software Technology*, Vol. 42, No. 2, pp. 117-128.
- [18] Gutwin, C. and Greenberg, S., A Descriptive Framework of Workspace Awareness for Real-Time Groupware, *Computer Supported Cooperative Work*, vol. 11 no. 3, p.411-446, 2002.
- [19] Helland T, A service oriented approach to software process support, MSc Thesis, Department of Computer Science, University of Auckland, 2004, 216pp.
- [20] IBM Corp, Rational Rose XDE Modeler, <http://www-306.ibm.com/software/awdtools/developer/modeler/>
- [21] Kaiser, G.E. Dossick, S.E., Jiang, W., Yang, J.J., Ye, S.X. (1998): WWW-Based Collaboration Environments with Distributed Tool Services, *World Wide Web*, vol. 1, no. 1, 1998, pp. 3-25.
- [22] Khaled, R., McKay, D., Biddle, R. Noble, J. and Tempero, E. (2002): A lightweight web-based case tool for sequence diagrams, *Proc. of SIGCHI-NZ Symposium On Computer-Human Interaction*, Hamilton, New Zealand, 2002.
- [23] Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in *Proceedings of CAiSE'96*, LNCS 1080, Springer-Verlag, Crete, Greece, May 1996.
- [24] Kim, C.H., Hosking, J.G. and Grundy, J.C. A Suite of Visual Languages for Statistical Survey Specification, In *Proceedings of the 2005 IEEE Conference on Visual Languages/Human-Centric Computing*, Dallas, Texas, 20-24 September 2005, IEEE CS Press.
- [25] Klein, P. and Schürr, A. Constructing SDEs with the IPSEN Meta Environment , in *Proc. 8th Conf. on Software Engineering Environments*, 1997, pp. 2-10.
- [26] Krasner, G.E. and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *JOOP*, vol. 1, no. 3, pp. 26-49, Aug. 1988.
- [27] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: Composing Domain-Specific Design Environments, *Computer*, 44-51, Nov, 2001.

- [28] Liu, N., Hosking, J.G. and Grundy, J.C. A Visual Language and Environment for Specifying Design Tool Event Handling, In *Proc. VL/HCC'2005*, Dallas, Sept 2005.
- [29] Liu, N., Grundy, J.C. and Hosking, J.G. A Visual Language and Environment for Composing Web Services, In *Proc. 2005 IEEE/ACM Int. Conf. on Automated Software Engineering*, Long Beach CA, Nov 7-11 2005.
- [30] Lyu, M. and Schoenwaelder, J. (1998): Web-CASRE: A Web-Based Tool for Software Reliability Measurement, *Proc. of International Symposium on Software Reliability Engineering*, Paderborn, Germany, Nov, 1998, IEEE CS Press.
- [31] Maurer, F., Dellen, B., Bendeck, F, Goldmann, S., Holz, H., Kötting, B., Schaaf, M. (2000): Merging project planning and web-enabled dynamic workflow for software development, *IEEE Internet Computing*, May/June 2000.
- [32] McIntyre, D.W., Design and implementation with Vampire, *Visual Object-Oriented Programming*. Manning Publications, Greenwich, CT, USA, 1995, Ch 7, 129-160.
- [33] Mackay, D., Biddle, R. and Noble, J. (2003): A lightweight web based case tool for UML class diagrams, *Proc. of the 4th Australasian User Interface Conference*, Adelaide, South Australia, 2003, Conferences in Research and Practice in Information Technology, Vol 18, Australian Computer Society.
- [34] McWhirter, J.D. and Nutt, G.J. Escalante: An Environment for the Rapid Construction of Visual Language Applications, *Proc. VL '94*, pp. 15-22, Oct. 1994.
- [35] Mehra, A., Grundy, J.C., Hosking J.G., A generic approach to supporting diagram differencing and merging for collaborative design, *2005 IEEE/ACM Automated Software Engineering*, Long Beach CA, Nov 2005.
- [36] Microsoft Visual Studio 2005: Domain-Specific Language Tools, <http://msdn.microsoft.com/vstudio/DSLTools/>, accessed 13 October 2006.
- [37] Minas, M. and Viehstaedt, G. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, *Proc. VL '95*, 203-210 Sept. 1995.
- [38] Myers, B.A., The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE TSE*, vol. 23, no. 6, 347-365, June 1997.
- [39] Object Management Group, Unified Modeling Language UML Resource Page, <http://www.uml.org/>, accessed 2 October 2006.
- [40] Phillips C, Adams S, Page D, Mehandjiska D, The Design of the Client User Interface for a Meta Object-Oriented CASE Tool, *Proc TOOLS*, 1998 Melbourne, p156-167.
- [41] Rekers, J. and Schuerr, A. Defining and Parsing Visual Languages with Layered Graph Grammars, *Journal Visual Languages and Computing*, vol. 8, no. 1, pp. 27-55, 1997.
- [42] Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. 1996. Designing object-oriented synchronous groupware with COAST, *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM Press, November 1996, pp. 21-29
- [43] Stoeckle, H., Grundy, J.C. and Hosking, J.G. Approaches to Supporting Software Visual Notation Exchange, *Proc HCC'03*, Auckland, New Zealand, Oct 2003, IEEE, 59-66
- [44] Vlisides, J.M. and Linton, M., Unidraw: A framework for building domain-specific graphical editors, in *Proc. UIST'89*, ACM Press, pp. 158-167.
- [45] Welch, B. and Jones, K. *Practical Programming in Tcl and Tk*, 4th Edition, Prentice-Hall, 2003.
- [46] Younas, M.a.I., R. Developing Collaborative Editing Applications using Web Services. *Proc. 5th Int. Workshop on Collaborative Editing, Helsinki*, Finland, Sept 15, 2003.
- [47] Zhang, K. Zhang, D-Q. and Cao, J. Design, Construction, and Application of a Generic Visual Language Generation Environment", *IEEE TSE*, 27,4, April 2001, 289-307..
- [48] Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, In *Proc. VL/HCC 2004 Rome*, Italy, 25-29 September 2004, IEEE CS Press, pp. 254-256.

2.4 Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications

Grundy, J.C., Hosking, J.G., Li, N., Li, L., Ali, N.M., Huh, J. Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications, *IEEE Transactions on Software Engineering*, vol. 39, no. 4, April 2013, pp. 487 - 515

DOI: [10.1109/TSE.2012.33](https://doi.org/10.1109/TSE.2012.33)

Abstract: Domain-specific visual languages support high-level modeling for a wide range of application domains. However, building tools to support such languages is very challenging. We describe a set of key conceptual requirements for such tools and our approach to addressing these requirements, a set of visual language-based metatools. These support definition of metamodels, visual notations, views, modeling behaviors, design critics, and model transformations and provide a platform to realize target visual modeling tools. Extensions support collaborative work, human-centric tool interaction, and multiplatform deployment. We illustrate application of the metatoolset on tools developed with our approach. We describe tool developer and cognitive evaluations of our platform and our exemplar tools, and summarize key future research directions.

My contribution: Co-developed initial ideas for the approach, co-led design of the approach, wrote initial software for the approach, co-supervised research assistant and 3 PhD students working on project, oversaw evaluation of the platform, wrote substantial amounts of the paper, co-lead investigator for funding for the work from Foundation for Research Science and Technology

Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications

John Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh and Richard Lei Li

Abstract—

Domain-specific visual languages support high-level modeling for a wide range of application domains. However, building tools to support such languages is very challenging. We describe a set of key conceptual requirements for such tools and our approach to addressing these requirements, a set of visual language-based meta-tools. These support definition of meta-models, visual notations, views, modeling behaviours, design critics and model transformations and provide a platform to realize target visual modeling tools. Extensions support collaborative work, human-centric tool interaction, and multi-platform deployment. We illustrate application of the meta-toolset on tools developed with our approach. We describe Tool Developer and cognitive evaluations of our platform and our exemplar tools, and summarise key future research directions.

Index Terms—



1. Introduction

Software Engineers use models to describe software requirements, design, processes, networks, tests, configurations and code. Construction, Engineering and Computer Systems professionals use models representing structures, plant, plumbing/electrics, materials, VHDL, electromagnetics, and processes/tasks. Health professionals have models for patient diagnoses, treatments and imaging. Business, Finance and Economics professionals use models to design and monitor processes/workflow. Families and friends may use models for family trees or to establish social networks. Our interest has been in the use of Domain-Specific Visual Languages (DSVLs) in these widely varied domains to assist domain users to better work with their complex domain models. A DSVL has a meta-model and visual notation allowing domain users to express complex models in one or more visual forms. Often, multiple visual forms are used to represent overlapping parts of the meta-model. Ideally DSVLs afford a “closeness of fit” to the Tool Developer’s problem domain [23, 65].

Working with models involves authoring, visualising, navigating, transforming, understanding, managing, and evolving models. There is a demand for appropriate, usable, scalable, sharable, robust and extensible tools to support these processes. Often multiple domain users must work together to author and review visual models. They sometimes want to model or access models in varying notations or interfaces, e.g. web or mobile device. They want effective tools to support the use of these DSVLs. However, building such tools is very challenging with the need for multi-view, multi-notational, and multi-user support, the ability for non-programmer Tool Developers to (re-)configure specifications while in use, and an open architecture for tool extension and integration.

Current approaches to constructing DSVL tools suffer from a range of deficiencies. These include limited domain targets, the need to use complex APIs and code for developing even simple environments, and a complex edit-compile-run cycle for reflecting even minor changes. These deficiencies provide barriers to use and typically prevent domain users and even developers from producing suitable DSVL tools. Visual specification approaches, compared to writing custom code, have shown their advantages in minimising design and implementation effort and improving understandability of programs [14,17,23,27,29]. This suggested to us that a visual language approach to support DSVL definition is likely to be a positive approach for the design and construction of domain-specific modelling environments, both for domain modelling tool users but also potentially for tool developers.

Manuscript received Feb 2011, Revised Aug 2011. Revised Jan 2012. Accepted April 2012.

Grundy is with Centre for Computing and Engineering Software Systems, Swinburne University of Technology, PO Box 218, Hawthorn, Victoria 3122, Australia (Email: jgrundy@swin.edu.au)

Hosking is with College of Engineering and Computer Science, Australian National University, Canberra, ACT 0200, Australia (Email john.hosking@anu.edu.au)

Huh is with Computer Science Department, University of Auckland, Private Bag 92019, Auckland, New Zealand (E-mail: designersheep@gmail.com)

Ali is with Faculty of Computer Science and Information Technology, Universiti Putra Malaysia (Email: hayati@fsktm.upm.edu.my)

Karen Li is with SolNet Solutions Ltd (Email: Karen.Li@solnetsolutions.co.nz)

Richard Li is with Beefand Lamb New Zealand Ltd (Email: Richard.Li@beeflambnz.com)

We focus on our conceptual/theoretic contributions to DSVL tool development and their realisation in this paper. We describe the key motivation for this research (Section 2) and survey related work (Section 3). We then provide an overview of our meta-tool approach (Section 4) by describing its key features and conceptual foundation. We then describe our integrated set of meta-DSVLs with a focus on novel behaviour specifications (Section 5), critic authoring and model transformation (Section 6). These are followed by a set of human-centric elements supporting accessibility and collaboration (Section 7) and then platform realisation (Section 8). We describe evaluation of our approach via exemplar DSVL tool development and a variety of more formal evaluations (Section 9) then provide discussion (Section 10) and conclude with a summary of key contributions from this research (Section 11).

2. Motivation

Software engineers use a range of models to describe software systems at various levels of abstraction. Some are very general and can be used to describe a wide range of software system characteristics. Some examples of such general-purpose visual modelling languages include the Unified Modelling Language (UML), Architectural Description Languages (ADLs), Entity-Relationship (ER) and Dataflow Diagrams (DFDs), and State Charts (SDs). This is akin to general purpose programming languages like C, C++, Java etc, compared to domain-specific languages (DSLs) for special purpose domains e.g. ATL for data transformation, BPEL for process orchestration and the Ant build scripting language. Many tools have been developed to support these general-purpose modelling languages. As they are general-purpose, investment in bespoke visual modelling tools for these modelling languages has generally been worthwhile.

Domain-specific visual languages (DSVLs) are more limited-domain modelling languages intended for modelling of limited classes of software systems or for modelling narrow aspects of systems, as DSLs are more limited, special-purpose forms of textual languages. One of the more successful DSVLs is the LabView visual language and environment [38], designed for instrumentation engineers configuring software and hardware interfaces in their domain. Other examples include various process modelling languages e.g. BPMN [75], load modelling languages e.g. Form Charts [19], component and service composition [25], and visual modelling languages for specific software application domains, such as health care plans [43], business workflow [51] and statistical surveying [44]. As these have much more limited purpose and scope, large investment in developing sophisticated tools for such domains is sometimes hard to justify or afford.

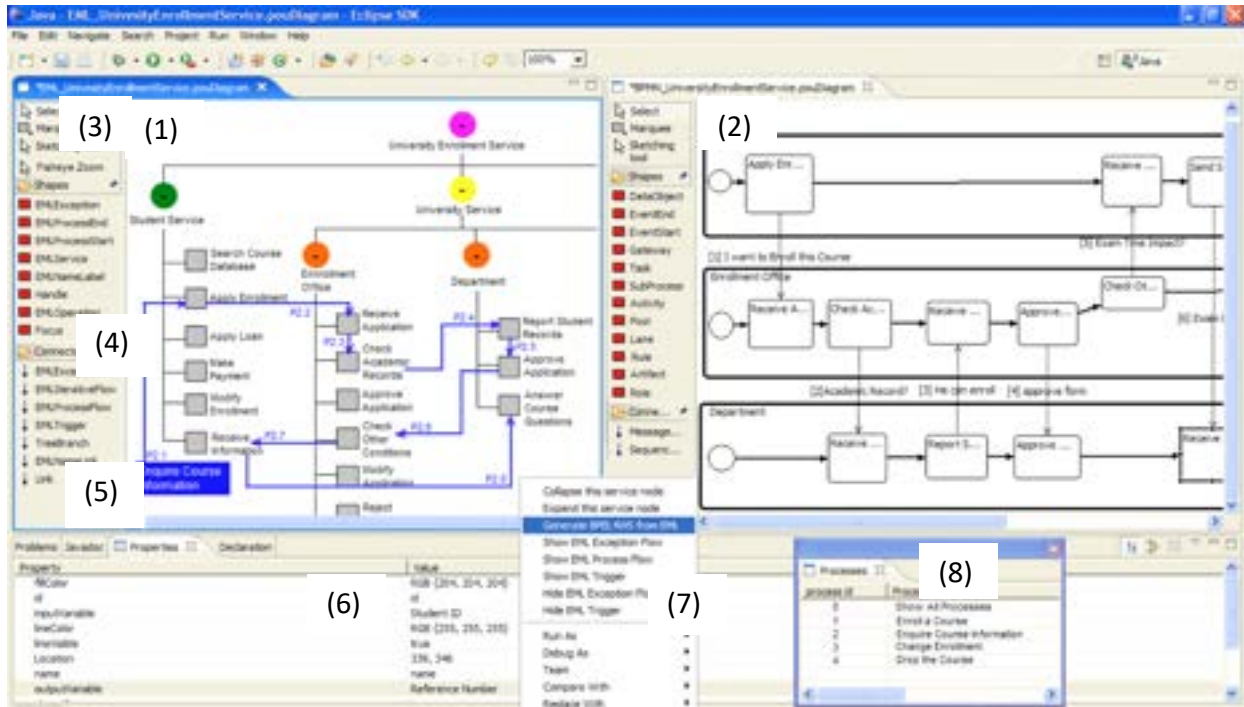


Figure 1. MaramaEML diagrams: (a) tree-based servicespecification with process overlay; and (b) BPMN process flow.

Consider an environment to support business service and process modelling, such as our MaramaEML domain-specific visual language tool [51]. Two DSVL examples from this tool are shown in Figure 1. MaramaEML needs to provide users with various visual notations for representing and understanding organisational services and process flows (Figure

1 (1)). In this example, the Enterprise Modelling Language (EML) DSVL is used in Figure 1 (1) to model a University student management application business processes. Other DSVLs may also be used, such as the Business Process Modelling Notation (BPMN), an emerging standard for process modelling for both engineers and business analysis. Figure 1 (2) shows a BPMN process model for the “enrole student” business process. Such DSVL tools require sophisticated diagram editing tool features (Figure 1 (3-5)), detailed property editing (Figure 1 (6)), script and code generators, such as a BPMN to BPEL generator (Figure 1 (7)), and various analysis tools, such as consistency checking of process models (Figure 1 (8)). Developing such DSVL tool features in e.g. Eclipse or Visual Studio is very time-consuming, requires detailed knowledge of many platform APIs, requires significant coding and debugging, and are difficult to maintain. As an example, to develop the EML tool in Eclipse using EMF, OCL and GEF projects to produce only a basic visual editor, approximately 2,500 lines of code need to be produced (ignoring the auto-generated EMF data structure code), roughly 2,100 implementing the GEF graphical editor and 400 implementing various meta-model constraints and behaviour. Use of GMF supports generation of around 1,800 lines of the graphical editor and additional controller code. The tool developer still needs to intimately understand and use Eclipse GEF, GMF, EMF, and OCL APIs, along with the different Event notification and Command frameworks, the XMI serialiser, the Eclipse plug-in and parts models, the OSGi-based plug-in configuration and deployment tool, and complex inter-dependencies between all of these.

As developing such DSVL environments is such a complex task it is generally exclusive to experienced software developers. Ideally we want this process to be simplified by leveraging meta-tool capabilities. Given many DSVLs may be useful and used by non-technical Tool Developers, ideally we want non-programmer Tool Developers to be able to develop visual notations and tools of relevance to their domain using their own domain knowledge. Thus our key goals include: 1) making DSVL tool implementation easier for experienced domain modellers (who may not always be experienced software developers), and users familiar with basic modelling concepts e.g. EER, OCL and meta-models; 2) allowing users to construct basic DSVL tools within one day, plus time for additional complexity such as backend code generators; and 3) leveraging the strength of the Eclipse platform, as our previous efforts with our standalone Pounamu [79] left us with infrastructure support needs that were too large and an inability to integrate seamlessly with other work.

Sutcliffe provides a useful conceptual framework for design modeling tools [73], shown in Figure 2. He argues that tools for model management should provide suitable model authoring (drawing) tools (1); simulation and prototyping support (2); design critics (3), knowledge reuse (4) and visualization (5) support; and annotation (6) and collaboration support (7).

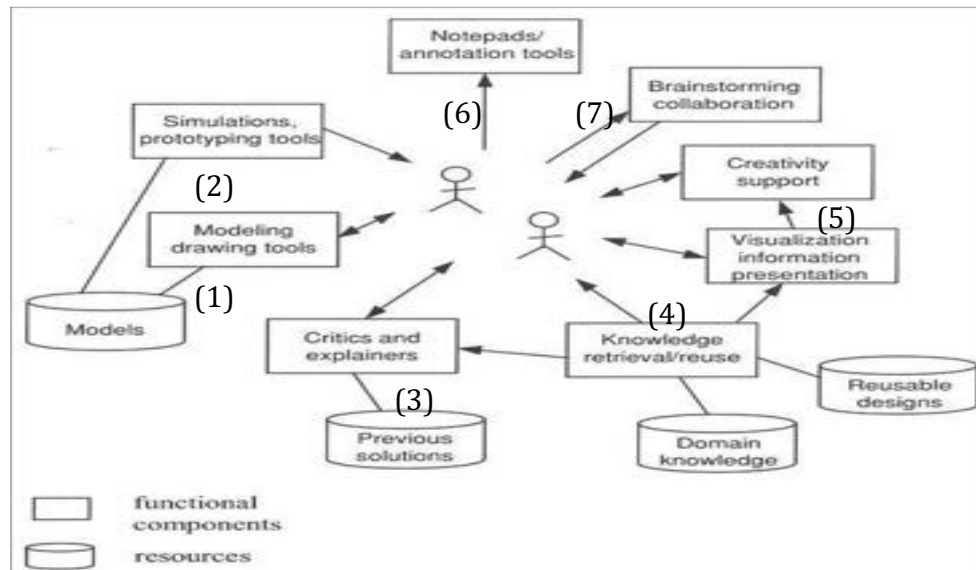


Figure 2. Sutcliffe's Design metadomain model, from [73]

Using Sutcliffe's conceptual model and our own experiences developing previous meta-tools, we have identified some key requirements for a meta-tool to construct DSVL tools as per our goals established above:

- Specifying modelling elements (Requirement 1) – these comprise (i) meta-model(s) representing canonical model(s) of domain-specific information, including entities and relationships, in the target DSVL; (ii) visual elements, made up of various icons and connectors, which form the visual representation(s) of domain models and elements; and (iii) diagrammatic views, which comprise notational elements for each DSVL diagram type and view-to-model mappings for the management of view-model consistency.

- Specifying modelling behaviours (Requirement 2) – these include dynamic and interactive tool effects such as event and constraint handling for both model and view manipulations and automated operations or processes.
- Model critiquing and transformation (Requirement 3) – to support proactive feedback on model quality and the exchange of view and model information with other tools, and backend code generation.
- Human-centric modelling (Requirement 4) - including scalable, sharable, usable, and intelligent support for collaborative and sketch-based editing and review.
- Modelling platforms (Requirement 5) – leveraging existing IDE facilities and related tools and making models available to domain users in appropriate ways.

3. Related Work

Three main approaches exist for the development of visual, multiple view DSVL environments: reusable class frameworks, diagram generation toolkits, and meta-tools.

General-purpose graphical frameworks provide low-level yet powerful sets of reusable facilities for building diagramming tools or applications. These include MVC [45], Unidraw [74], COAST [72], HotDoc [13] and Eclipse’s GEF [1]. While flexible and powerful, these frameworks typically lack abstractions specific to multi-notation and multi-view visual language environments. Thus construction of DSVL tools is very time-consuming. For example, supporting multiple views of a shared model in GEF requires significant programming effort. Given the challenge, a variety of special purpose frameworks for building multi-view diagramming tools have been developed. These include JViews [30], IBM ILOG JViewsDiagrammer [3] and NetBeans Visual Library [6]. These offer reusable facilities for visual language-based environments, but still require detailed programming and a edit-compile-run cycle, limiting their ease of use for exploratory development and for tool developers without detailed technical knowledge.

A number of more targeted diagram generation toolkits have been produced to make DSVL tool development easier. These include Vampire [60], VisPro [77], JComposer [30], PROGRES [70], DiaGen [62], VisualDiaGen [63], Merlin [4] and VEGGIE [78]. All of these diagram generation toolkits use code generation from high-level specifications. Some use meta-models and associated editor characterisation as their source specifications e.g. JComposer and Merlin (using EMF model). Others, e.g. DiaGen, VisualDiaGen, PROGRES, VEGGIE and VisPro, use formalisms such as graph grammars and graph rewriting for high-level syntactic and semantic specification of visual tools. Code generation approaches suffer from similar problems to many toolkits: often requiring an edit-compile-run cycle and difficulty in integrating third party solutions. Tool developers sometimes have to resort to code for some editor capabilities (e.g. customisation of shapes in Merlin), or cannot add some desired support features to their target tools. Formalism-based visual language toolkits typically limit the range of visual languages supported and are often difficult to extend in unplanned ways. They also require understanding of the formalism, most commonly graph grammars that specify a set of valid transformations from one graph state to another. Historically graph-grammar based visual editors had more limited editing capabilities than toolkit or framework coded tools, requiring valid graph transforms only and menu-driven editing approaches. More recent toolkits such as DiaGen provide graphical editors with growing flexibility.

A third approach to realising DSVL tools is using meta-tools. Meta-tools provide a purpose-designed IDE for generation, configuration and exploratory development of other tools. These include KOGGE [20], MetaEdit+ [41], MOOT [67], GME [47], Pounamu [79], TIGER [21], DiaMeta [64], Eclipse Graphical Modelling Framework (GMF) [2], AToM³ [46] and Microsoft DSL Tools [5]. Meta-tools for DSVL environments typically provide separate specifications of different tool aspects. High-level definitions of tool metamodels, shapes, connectors, and mappings from metamodel elements to shapes and connectors are used to effectively generate a tool structure implementation. While a popular mechanism has been provided to allow DSVL tools to be quickly generated and configured with minimal user specification effort, features such as complex editing behaviour, tool integration, code generation, design critics, human-centric editing and accessibility are generally not directly supported and must be hand-coded if required. Typical distinctions of the existing meta-tools include the paradigms used for metamodelling (e.g. UML used by DSL Tools, EMF used by GMF, ER used by Pounamu, and graph grammars used by TIGER), the variety of facilities offered for visual notation design (e.g. drawings, forms, templates and abstract specifications), the directness of multiple view specification support (e.g. automated, via visual mapping, or with the need for substantial coding), and of particular concern to us, the abstraction level of DSVL tool behavioural specification support (e.g. using scripting languages, building blocks provided in the meta-tool frameworks or visual notations). Microsoft DSL Tools provides an integrated visual specification editor with designated parts for metamodel, visual notations and their representational mappings. Multiple, linked views are however not directly supported and realising them in generated tools requires much coding and configuration. Model validation, diagramming rule definition and artefact generation are well supported in the SDK through the use of framework APIs, metadata attributes (annotations), object-oriented inheritance and template-based generators. Reuse and integration are also well supported leveraging Visual Studio’s extensibility. However, these are not raised to a visual abstraction level. Pounamu provides built-in support and a form-based mapper for multiple, linked views. It also allows some limited form of visual

event handling behaviour specification. DiaMeta allows rule-based diagram layout control behaviour to be specified using metamodels in EMF. ATOM³ provides a DSVL (called SLAMMER [31]) for the use of generalised visual patterns for DSVL model measurement and redesign. MetaEdit+ provides common rules for Tool Developers to choose/adapt, and automatically delivers them in model instances. For code generation and integration, advanced built-in scripting commands are used. GMF supports live validation of diagrams. It also allows models to be specified in OCL leveraging the EMF validation framework. C-SAW [76] in GME supports model constraint specification in OCL using a separate form-based editor view. Most meta-tools aim for a degree of round-trip engineering of the target tools. Typically they provide support for their target domain environments and tool developers may need little technical programming ability to use the meta-tools. However many are limited in their flexibility in terms of target tool capabilities able to be specified and their integration with other tools. Pounamu and IPSEN are good examples of these limitations, and additionally incur large effort to build features provided by general-purpose IDEs. Thus a recent trend has been to build both diagram editor toolkits and meta-tools on top of existing IDEs such as Eclipse and Visual Studio. These can then leverage save/load resource management features, modelling frameworks, graphical and text editor frameworks, a common user interface look and feel, and many third party extensions.

Considerable research has gone into providing proactive critiquing support in software IDEs, DSVL tools and other design-oriented domains. These give proactive support to users as they model. Good examples include design advice in ArgoUML and advice on Java programming constructs in Java Critiquer [69, 71]. However, no high-level critic definition approaches have been incorporated into DSVL meta-tools. Model transformation and code generation support has been recognised as a critical feature for many DSVL tools employed for model-driven engineering problems. This has included approaches such as enterprise data mapping, GXL and VMETS [11, 34, 48]. Typically these approaches have been standalone model transformation or exchange tools rather than integrated DSVL meta-tool support features. Providing more human-centric editing capabilities for DSVL tools has also been recognised in much recent research. A number of approaches to provide sketching-based support have been developed, such as extensions to DiaGen for sketch-based recognition [12]. Diagram differencing, merging and collaborative editing support, have been explored by Mehra et al. [61] and Lin et al. [52]. Most of these approaches provide fixed, closed groupware functionality, however, and we desired more flexibility over target visual design tool collaborative work facilities. In addition, we wanted these capabilities to support integration with existing tools. Thin-client, or rich internet application interfaces, have been explored; examples include MILOS [59], a web-based process management tool, BSCW [9], a shared workspace system, Web-CASRE [56], which provides software reliability management support, web-based tool integration, and CHIME [39], which provides a hypermedia environment for software engineering. Most of these provide conventional, form-based web interfaces and lack web-based diagramming tools. Recent efforts at building web-based diagramming tools include Seek [42], a UML sequence diagramming tool, NutCASE [22], a UML class-diagramming tool, and Cliki [57], a thin-client meta-diagramming tool. All have used custom approaches to realise thin-client diagramming. They also provide limited tool tailorability by Tool Developers and limited integration support with other software tools. Many are standalone efforts whereas ideally meta-tools should be able to support these capabilities for all target DSVL tools.

In summary, a number of approaches have demonstrated the capability to capture domain model elements using high-level specifications (our Requirement 1). However, the majority require detailed programming and framework knowledge, or understanding of complex formal information representation models (e.g. graph grammars), for DSVL tool development. Few have managed to support behaviour specifications accessible to non-programmer Tool Developers (Requirement 2). There is to date limited support for DSVL tool critic editing based on knowledge (best practices) reuse and for model transformations (Requirement 3). Few approaches support simple, live, evolutionary, and collaborative development of DSVL tools with good accessibility and the use of Tool Developers' own domain knowledge (Requirement 4). More approaches are leveraging IDEs to realise target DSVL tools but these still lack high-level support for tool integration (Requirement 5).

4. Overview of Our Approach

We wanted to simplify the DSVL tool development process by extending meta-tool capabilities. Our approach is to generate DSVL tools from a variety of high-level, visual specifications in a meta-tool, called Marama. Figure 3 relates our key requirements for DSVL meta-tools from Section 2 above to Sutcliffe's Design metadomain model [73].

Our Marama approach addresses most of the components in the Design metadomain, providing strong support in some areas and partial in others. Our core approach is a set of visual, declarative specifications of domain meta-models, their visual representations and their views. These include an extended entity-relationship (EER) modeller for the specification of domain specific meta-models (defined using a "Meta-model Designer"); a WYSIWYG "compose-and-edit" approach to visual notational element specification (the "Shape Designer"); and mapping tool for the filtering of model elements into (possibly multiple) views (diagrams) and the facilitation of consistent view and model editing (the "View Type Designer"). These collectively address our key Requirement 1.

Several approaches are used to specify advanced DSVL tool behaviours, our key Requirement 2. One is a visual, declarative specification of constraints on models that augments the Meta-model Designer. A form of OCL constraints is supported by visual representations of model element dependency and a spreadsheet-like model/formulae metaphor. A visual, imperative specification of event-based behaviour provides a set of “Event Handler Designers”. A declarative augmentation of shape designs and view type designs is used to specify (limited) automatic layout functionality. A high-level architectural component model allows Tool Developers to specify intra- and inter-tool communication and co-ordination. These all make use of an integrated event handling behavioural model based on event-condition-action rules. Together, these meta-tools provide strong support for defining models, visual notations for models, and drawing tools for authoring visual notations of models. They provide basic support for information visualisation, visual debugging and simulations in target DSVL tools.

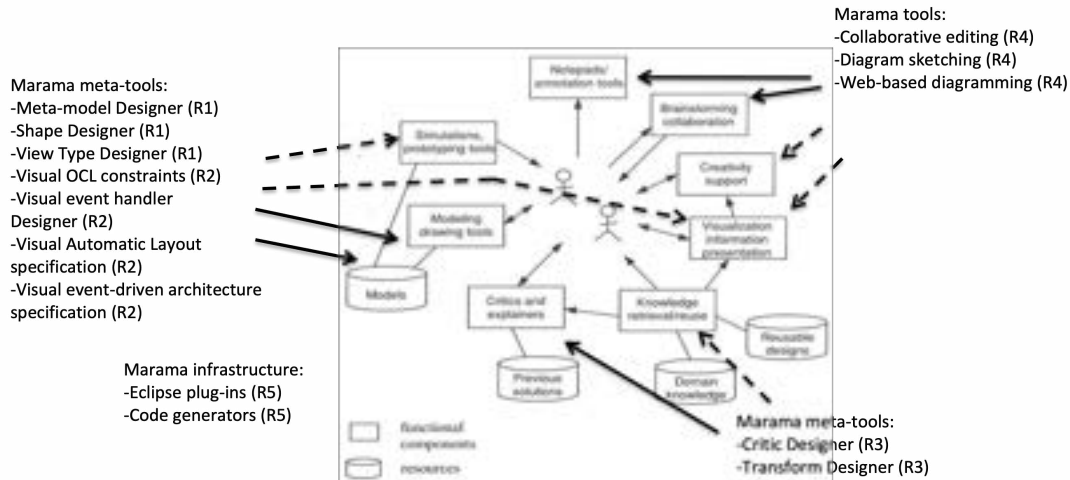


Figure 3. Relating Sutcliffe's model to Marama capabilities.

A visual, template-based critique and feedback authoring system is used to specify a set of “design critics” for proactive advice to DSVL tool users (the “Critic Designer”). A visual, tree-based schema mapping approach facilitates model transformation, model import and code generation (the “Transform Designer”). Together these support a range of proactive design critics for a target DSVL tool and a range of information exchange, reuse and code generation facilities, addressing our key Requirement 3.

We have experimented with a range of ways of supporting human-centric modeling in Marama DSVL tools. These include a sketched-based overlay for diagram editing via intelligent ink annotations, support for multi-user collaborative editing, and thin-client, web-based interfaces for more accessible information presentation and authoring. Together these allow generated DSVL tools to support flexible annotation, collaboration and brainstorming. They provide some limited support for creative design and information visualisation and thus partially address our Requirement 4. We have realised Marama as a set of plug-ins using the Eclipse IDE, addressing our Requirement 5.

5. Meta-model and behavioural specifications

We use our MaramaEML business process modelling tool introduced in Section 2 as a running example of a complex DSVL tool to be specified and generated with Marama. Consider a tool developer wanting to develop such a tool to enable high-level business process modelling, BPEL script generation, integration of a third party LTSA model checker for BPEL, and information visualisation support. We first show how the basics of such a DSVL tool can be specified in Marama. Later we illustrate how we can augment the basic tool specification with design critics, model transformation and human-centric modelling support. We then describe our realisation of the Marama DSVL meta-tool using Eclipse and Microsoft DSL Tools IDEs.

5.1. Specifying DSVL structural aspects (Requirement 1)

Structural elements, including entities, relationships, shapes, connectors and view types, form the backbone of a DSVL tool specification. Defining domain specific concepts is a mind-mapping process of abstracting out entities, relationships, sub-typing, roles, attributes and keys as encountered in the Tool Developer domain. In Marama, this process is meta-modelling. The Marama Meta-model Designer tool, illustrated in Figure 4, uses an EER representation. We chose an EER approach, rather than MOF or UML, for simplicity for our target Tool Developer community, which includes non-

technical DSVL tool developers. The Marama EER meta-model can, however, be transformed to and from other meta-modelling representations. Figure 4 shows a basic meta-model for representing MaramaEML BPMN constructs.

In this example, the MaramaEML tool designer has specified a range of entities (green square icons) to represent fundamental domain concepts e.g. Activity, StartEvent, StopEvent, Gateway, Comment, Swimlane etc. Some of these have been generalised to super-types e.g. Element, Event and ProcessElement via generalisation relationships (unfilled arrow pointing to general type). This allows types to share common information including attributes and associations. Several associations have been specified, shown as pink oval rectangles e.g. Comment-to-Element, Activity-to-Activity etc. These connect elements via named association links. Multiple meta-model diagrams are possible to manage complexity.

The tool developer can also specify “event handlers” which detect editing events or provide tool users with pop-up menus to invoke additional functionality. In this example, a GenerateUniqueID event handler creates a unique ID upon the creation of new Elements. Event handlers are tool developer written Java-coded scripts, reused and parameterised scripts, or are generated by various other Marama visual specification meta-tools (see later). Marama provides a comprehensive set of APIs allowing tool developers to query and manipulate any aspect of Marama meta-model or diagram data structures. Event handler code can also access any Eclipse APIs used by Marama.

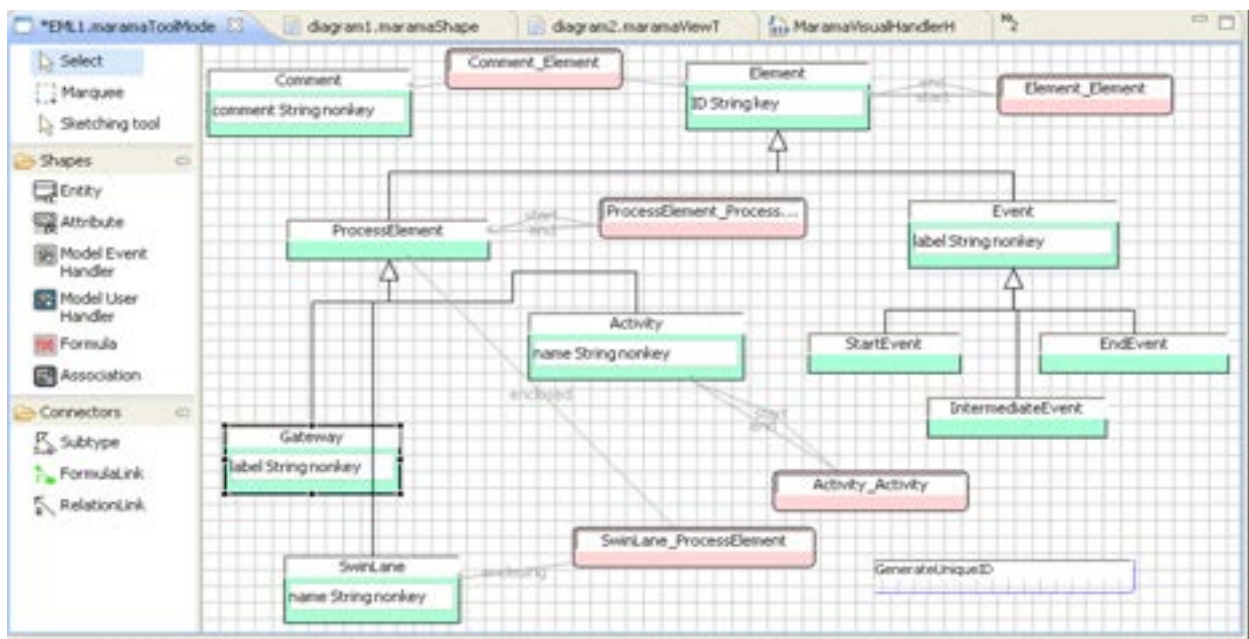


Figure 4. An example MaramaEML meta-model in the Marama meta-model designer tool.

A WYSIWYG Shape Designer tool, illustrated in Figure 5, allows rapid composition of icons and connectors for representing the domain specific concepts defined in the meta-model. Icons and connectors are specified in a generic, abstract form via drag-and-drop. This uses a semi-concrete representation of the specified shape and connector notation for immediate design feedback. In this example, Start and Stop event shapes (ovals); an Activity shape, a Group shape, a Comment shape and a Gateway shape have been designed. Some of the editable attribute values for the Activity shape specification are shown. Some associations are shown, including event flow, comment anchor and grouping.

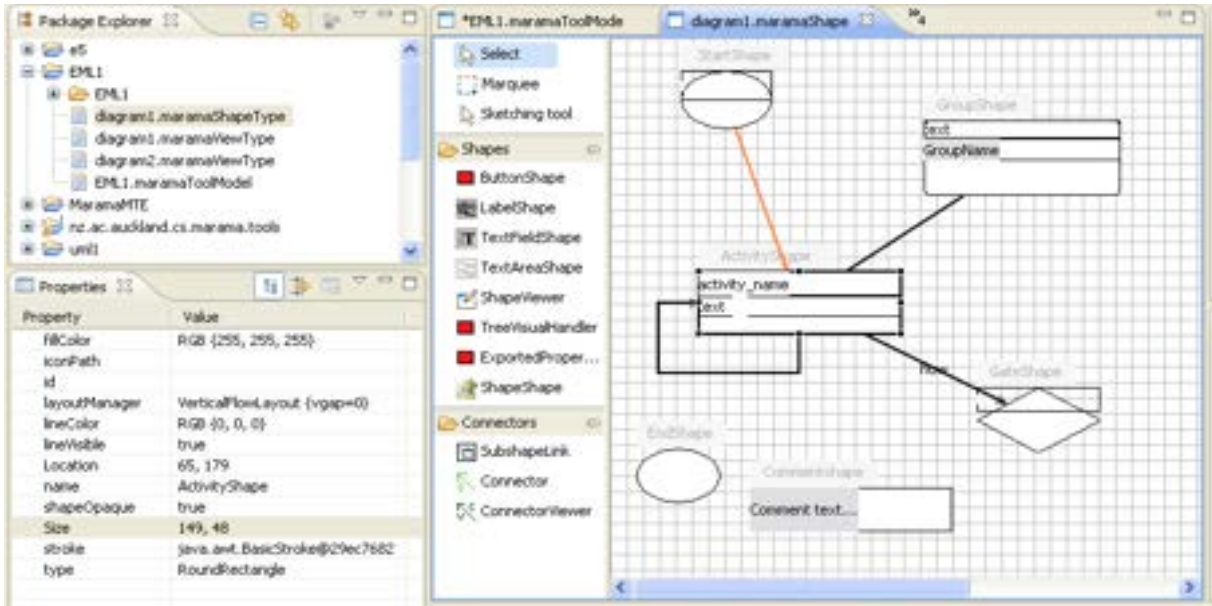


Figure 5. An example of some MaramaEML BPMN concrete notation shape and connector designs in the Marama Shape Designer.

The View Type Designer tool, illustrated in Figure 6, specifies which visual elements are included in a diagram (which we call a “view type”) and their relationships to the underlying model elements (including attribute mappings). In this example, the main BPMN view type is specified. This includes Activity, Group, Comment, Event and gateway shape mappings to appropriate meta-model entities, and mappings of various connectors such as Flow and Comment anchored to meta-model associations. Various attributes of shapes and connectors can be mapped to meta-model entity and association attributes. This implies automatic maintenance of a bi-directional consistency between realised view and model element attributes in models developed using the generated DSVL tool. A view type wizard is provided to select defined entity, association, shape and connectors and generate an initial view type. View types may also have event handlers defined. For example, in this example GroupContainsProcess and AlignContainedShapes are “event triggered” handlers for managing grouped BPMN process elements inside a Group shape. A generateBPEL event handler is a user-selected pop-up menu item that generates BPEL script from the BPMN meta-model elements.

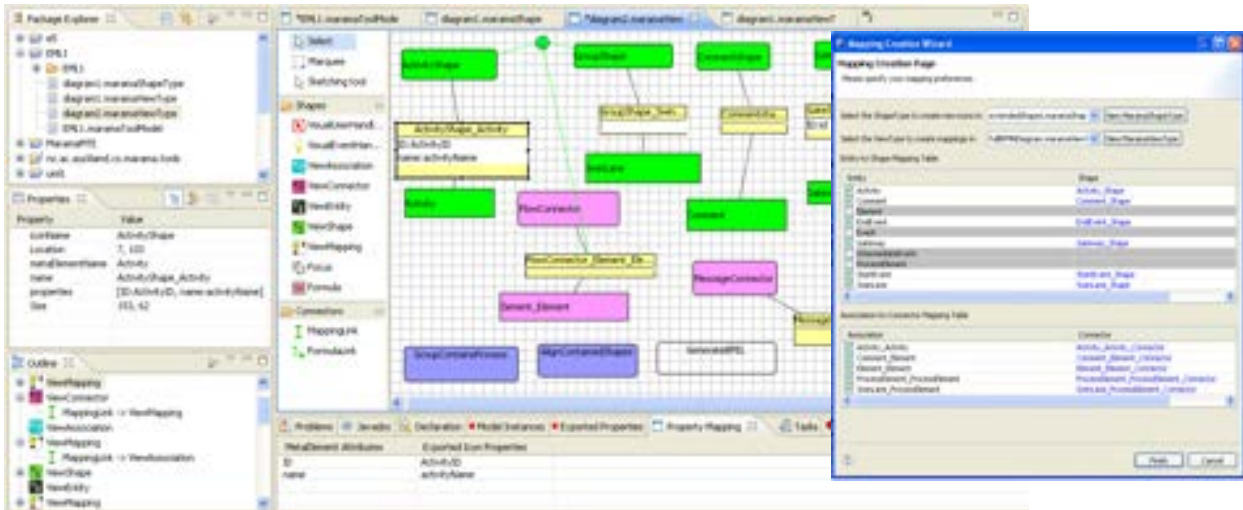


Figure 6. Example of Marama view type designer (left) and view type creation wizard (right).

Having specified a DSVL or set of DSVLs using these basic structural specifications (meta-model representations of domain concepts, their visual representations and view mappings), users can, with no further work, realise the specified tool and use it to create domain specific models and diagrams using the DSVL(s). A model project contains one model

instance with multiple view (diagram) instances, all kept consistent with one another. Both the model and view instances can be manipulated via user interactions. A view instance realises all the icon and connector types defined in the View Designer as Palette tools. As per the mappings of visual elements (icons and connectors) and model elements (entities and relationships) specified in the View Designer, an icon/connector instance in the view instance automatically generates a model entity/relationship, with appropriately mapped property values. Unmapped visual/model properties of a visual/model element are persisted independently. The model project contains a model file (file extension “.model”), which stores the runtime model state.

5. 2. Specifying DSVL Tool behavioural aspects (Requirement 2)

Developing modelling behaviours is a key challenge in DSVL tools. Many approaches require programming knowledge and access to complex APIs. Our Marama event handlers written in Java code are very powerful but suffer these same problems. However, behavioural specification from a high-level abstraction is generally difficult to achieve. Appropriately chosen metaphors are important for mapping a specification onto a user’s domain knowledge, not only for structure but also for behaviour specifications.

In Marama we chose a declarative spreadsheet-like metaphor to specify model level dependencies and constraints and an imperative dataflow-like Event-Condition-Action-based metaphor for view level event handlers. We use a subscribe-notify Tool Abstraction metaphor to describe event-based tool architecture and multi-view dependency and consistency. These three different metaphors were generalised from our earlier work on domain-specific event handling specification [53, 54, 55], integrated via a common model and unified user interface representation [50].

5. 2. 1. Declarative model constraint specification

A number of approaches separate model constraint specifications from their meta-model specifications [5, 41]. This causes potentially serious *hidden dependency* issues (one of the Cognitive Dimensions framework dimensions [23]). We believe constraints such as attribute value boundaries and dependencies, relationship multiplicities and cyclic reference checking can be presented more simply and clearly within the same meta-model specification by annotating existing contextual elements. MaramaTatau is an extension of Marama’s EER meta-model designer specifications adding declarative dependency/constraint specifications and high-level visual annotations. Value dependencies and modelling constraints are state-change events handled in Marama via a uni-directional change-propagation mechanism with side-effects to dependent components, the same approach used by formula evaluation in spreadsheets. However, we wished to minimise Cognitive Dimensions [23] tradeoffs such as hidden dependency and visibility issues between constraint and meta-model specifications that are common in spreadsheet like approaches. MaramaTatau supports visual construction of formulae to specify model structural dependencies and constraints at a type rather than the usual spreadsheet instance level. We chose OCL as the primary textual formula notation as: OCL expressions are relatively compact; OCL has primitives for common constraint expression needs; OCL is a standardised language; and the quality of OCL implementation is increasing.

In Figure 7 we have used MaramaTatau to extend the MaramaEML meta-model with a constraint specifying that a StartEvent must have at least one Activity connected to it. Small green circular annotations on an attribute or entity indicate that an OCL formula has been defined to calculate a value (eg the id of an Element) or provide an invariant (e.g. the cardinality constraint on a StartState). We use green arrowed lines to show formula dependency annotations and grey borders to annotate sensible elements to be involved in a formula construction at a certain stage. Formula construction can be done textually, via a Formula Construction (OCL) view, or “visually” by direct manipulation of the meta-model and OCL views to automatically construct entity, path, and attribute references and function calls. Clicking on an attribute places an appropriate reference to it into the formula.

In this example, the tool developers create a formula for the StartEvent entity (1). They then begin specifying the constraint using a combination of the visual elements in the meta-model and the formula editor view (2). They specify the StartEvent entity (self) by clicking on it (3), which highlights in the visual meta-model elements, associations and/or attributes that can next be validly added to the formula (4). In this example, the developers choose its named association “start”, linking StartEntity to the first Activity entity for the BPMN specification (5). Clicking on a relationship and then an attribute generates a path reference in the formula (self.start in this example). A difference from the spreadsheet approach is that only certain elements are semantically sensible at each stage of editing, whereas in spreadsheets, almost any cell may be referenced. The tool developers then specify the size() function, from the available functions list on the right (6), and add a constraint of $\langle 0$ (i.e. a StartEntity must have one or more connected Activity entities to be valid).

The cardinality constraint on the Service entity is thus specified by the OCL expression “self.start->size() $\langle 0$ ”. When this formula evaluates false for a StartState in a model instance, a constraint violation error is generated, with a problem marker representation appearing in the Eclipse Problems view to provide the user with details of the violated constraint. In this example, to solve the error, the user needs to connect the StartState entity to an Activity entity in the BPMN diagram. When this is done, the constraint evaluates to true and the constraint error is removed from the Problems view.

We have carefully defined interaction between the OCL and meta-model views to enhance visibility and mitigate hidden dependency issues. OCL and EER editors are juxtaposed, improving visibility, and simple annotation of the model elements indicates formulae related to them are present and semantically correct/incorrect. Formulae can be selected from either view so constraints can be readily navigated to/accessed. The dependency links permit more detailed understanding of a formula. The annotations are modified dynamically during editing for consistency. Dependencies are only made visible if a constraint is selected to minimise scalability issues and support task focus. The approach is similar to conventional spreadsheet dependency links but applied to graphical modelling. Constraints with formula errors are highlighted in red (7). A set of example constraint indicators and formulae are shown in the final example screen dump (8).

5. 2. 2. Declarative diagram constraint specification

Some specialised visual relationships in views, such as various composite icon layout mechanisms can also be expressed declaratively using the same formulaic approach. Instead of using Java event handler specifications, the tool developer can reuse some limited visual specifications. We have extended our OCL-based meta-model dependency/constraint specification technique by adding some reusable functions and applying them to iconic notations at the view level. Figure 8 (left) shows a visual constraint specified in the BPMN view type design of MaramaEML. The enclosure relationship (icon is contained in but can be moved within the enclosing icon) is specified between a GroupShape and ActivityShape, using the FlowConnector connector to manage this relationship i.e. all AcitivityShapes related to a GroupShape by a FlowConnector are “grouped” with that GroupShape. When the GroupShape is moved or resized, Marama automatically moves and resizes the grouped ActivityShapes, Figure 8 (right). Adding, moving, resizing or deleting ActivityShapes in the group may cause the GroupShape to be automatically resized/moved to continue to contain them. Other constraints that can be specified in this way include full containment with vertical or horizontal layout of contained shapes, shapes auto-aligned (“pinned”) to the edge of another shape, and shapes auto-located within another shape.

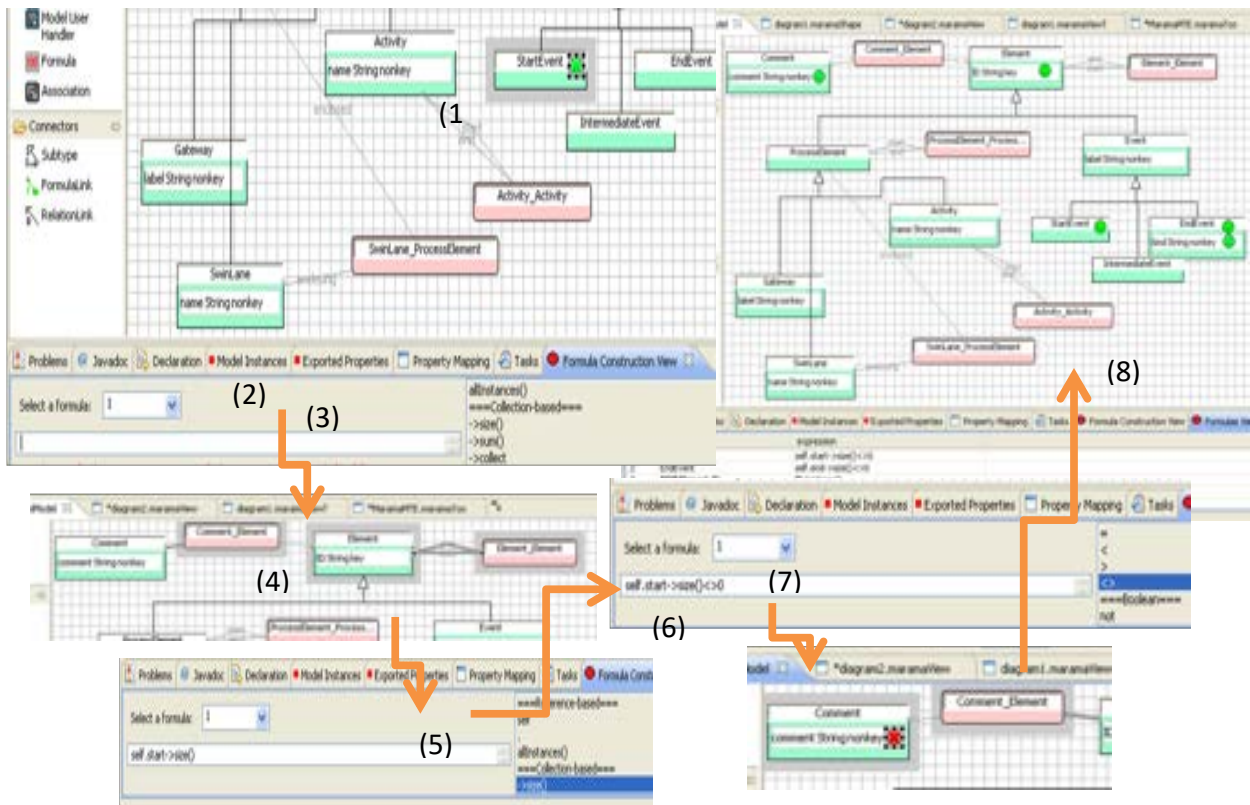


Figure 7. Examples of visual constraint specification using the MaramaTatau extensions to the Marama meta-model designer.

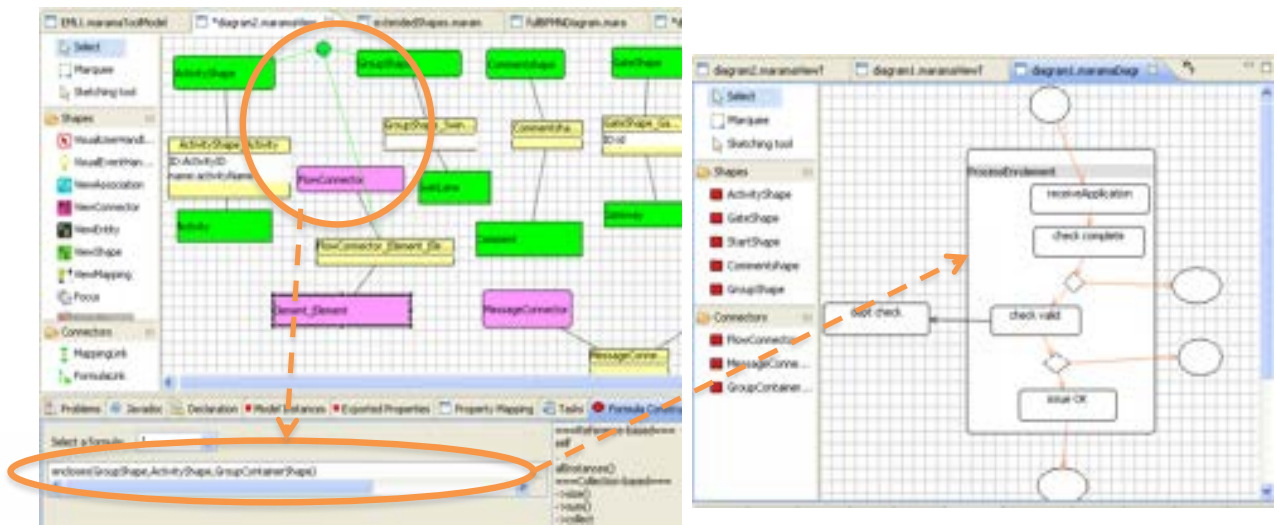


Figure 8.(a) example of an encloses() shape layout constraint being specified in the Marama view type designer and (b) its effect in the Marama generated BPMN editor.

5. 2. 3. Declarative diagramlayout specification

More complicated auto-layout using trees and force-directed layout is supported by further visual augmentation of Shape and view type specifications. MaramaALM (Automatic Layout Mechanism) is an extension to the shape and view type designers that allows shapes to be specified as participating in complex tree (vertical or horizontal) and/or automatic force-directed layouts [66]. Figure 9 (left) shows the tool designer augmenting shape designs in the Marama shape designer with visual annotations (small dark green octagon shapes). These indicate participation of the shape in automatic layouts in target diagrams. The augmented view type designer in Figure 9 (right) shows layout event handlers added to a view type that will use these augmented shape type specifications to apply tree and/or force-directed layouts in the target Marama design tool. The tool designer specifies the shapes that participate in the tree or force-directed layout, connectors used to link these related shapes, and configurations e.g. horizontal or vertical tree; auto-resize or not; and amounts to space and auto-relocate.

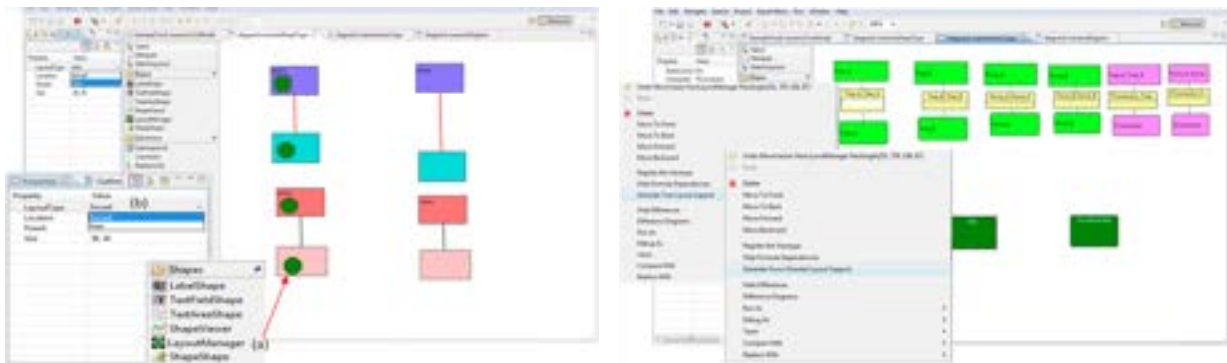


Figure 9. Specifying participation in auto-layouts (left) and type of auto-lout(s) to add to a view type definition (right).

Figure 10 shows the specified auto-layouts in use. In the top figure a diagram has hierarchical tree layout. The user can change this to a horizontal layout as shown on the right. Marama automatically re-lays out the diagram components for shapes participating in the tree. Figure 10 (bottom) shows a force-directed layout in use. The right-hand side shows the resizing and repositioning of shapes in the view that participate in the force-directed automatic layout specified above. Note MaramaALM supports combination of tree and force-direct layouts for the same diagram e.g. a horizontal tree lays out the oval shapes and a force-directed layout lays out the rounded rectangle shapes.

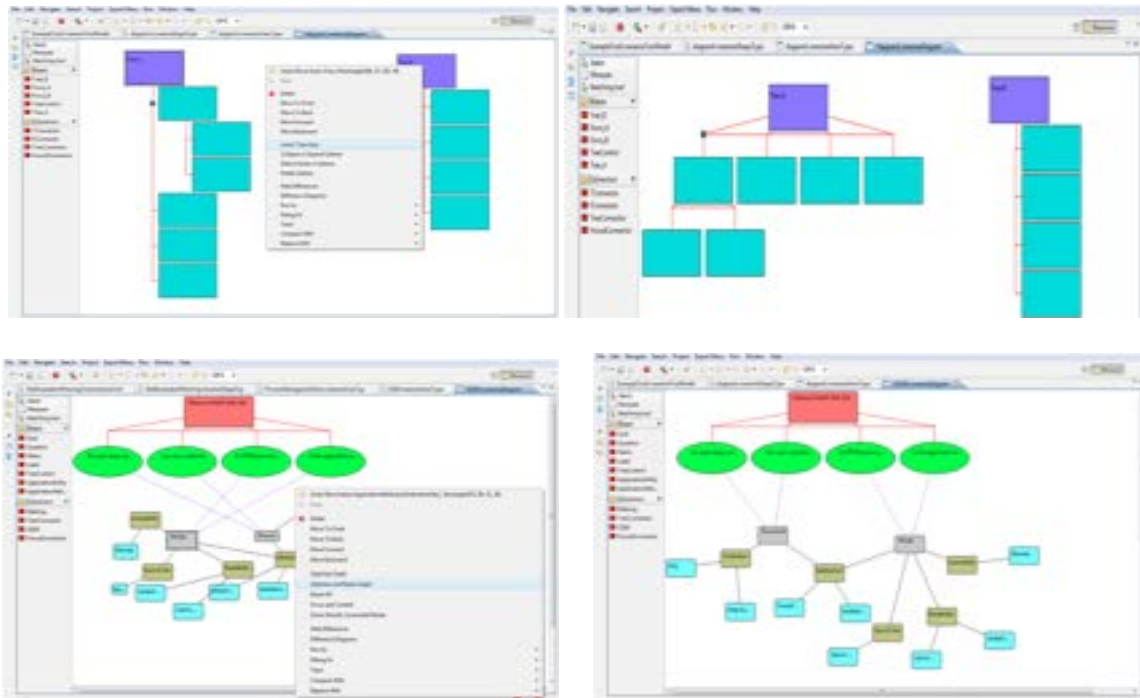


Figure 10. Tree layout (top) and force-directed layout and optimisation (bottom).

5. 2. 4. Visual event handler specification

Our declarative spreadsheet-like approach allows tool designers to specify constraints in a simple visual and declarative way. However, there are limitations with this approach. It is awkward to express more operational behaviours such as composite queries, filters and actions in event handling, and the flow of data between them. To address this, we developed a visual event flow language (Kaitiaki): an “Event-Query-Filter-Action (EQFA)” notation for expressing view level constraints/operations. The approach is based on our earlier Serendipity [27] visual event processing language. When constructing event handlers Tool Developers select an event type of interest; add queries on the event and Marama tool state (usually diagram content or model objects that triggered the event); specify conditional or iterative filtering of the event/tool state data; and then state-changing actions to perform on target tool state objects. Complex event handlers can be built up in parts, via sub-views, and queries, filters and actions can be parameterised and reused.

The visual language design focuses on modularity and explicit representation of data propagation. We have avoided abstract control structures and used a dataflow paradigm to reduce cognitive load. Key visual constructs are events, filters, tool objects, queries on a tool’s state, state changing actions plus iteration over collections of objects, and dataflow input and output ports and connectors. A single event or a set of events is the starting point. From this data flows out: event type, affected object(s), property values changed, etc. Queries, filters and actions have parameter bindings via data propagated through inputs. Queries retrieve elements from a model repository and output data elements; filters apply pattern-matching to input data, passing matching data on as outputs; actions execute operations which may modify incoming data, display information, or generate new events.

Queries and actions execute when input data are available (data push). If there are no input parameters, queries and actions trigger whenever parameters to a subsequent flow element have values (pull). We predefined a set of primitives for these constructs providing operations useful for diagram manipulation, by abstracting from a large set of diagram manipulation examples. These involve collecting, filtering, locating or creating elements, property setting, relocating/alignment, and connection. Multiple flows are supported. Concrete DSVL icons, specified in the shape designer, are also incorporated into the visual specification of event handling as placeholders for Marama tool state, to annotate and mitigate the abstraction, making the language more readable.

Figure 11 shows an event handler specified for aligning ActivityShape shapes. The handler responds to a Marama “shapeAdded” or “ShapeMoved” event (1). The concrete representations of Activity and Gateway shapes (2) filter the added and moved events to only these types of shapes. A further filter checks that the Activity and Gateway shapes are contained within a Group shape (3). All of the shapes within the GroupShape are then retrieved by a Query (4) and an Action aligns all of the grouped shapes (5).

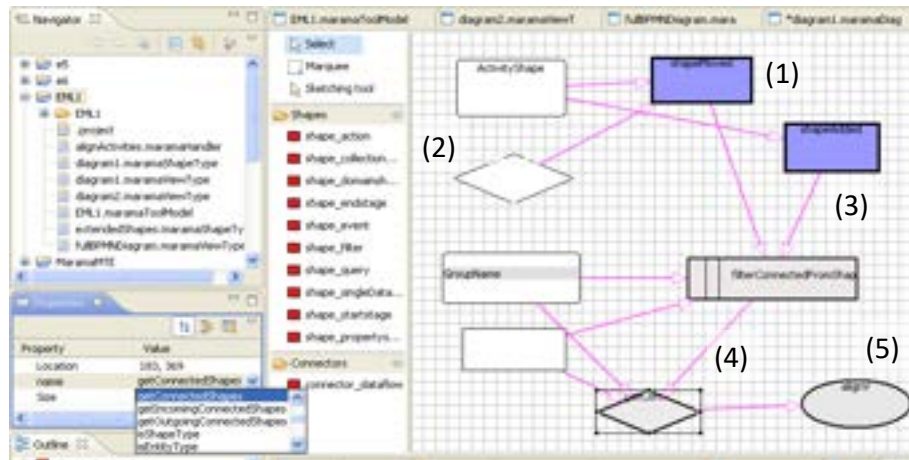


Figure 11. Specifying an event-driven shape alignment algorithm with the MaramaKaitiaki visual language.

5. 2. 5. Visual event-based architecture specification

The declarative and imperative constraint/event handling approaches described above are of low-to-medium abstraction and lack the ability to describe and affect the overall high-level architecture of a DSVL tool. We chose to use the Tool Abstraction (TA) [26] metaphor, with a notation designed for our ViTABaL-WS web service composition [53] work, to provide a view to describe event-based tool architecture. This mitigates multi-view dependency and consistency issues. TA is a message propagation-centric metaphor describing connections between “toolies” (behaviour encapsulations which respond to events to carry out system functions) and “abstract data structures” (ADSs: data encapsulations which respond to events to store/retrieve/modify data) that are instances of “abstract data types” (ADTs: typed operations/messages/events). Connection of toolies to other toolies and ADSs is via typed ports. TA supports data, control and event flow relationships in a homogeneous way, allowing a focus on architecture level abstractions and describing separated concerns including tool specific events, event generators and receivers, and responding behaviours such as event handlers. Key modelling constructs include event sources/sinks, Marama components, event handlers, toolies (data processing), ADSs (data management), data storage and error handlers.

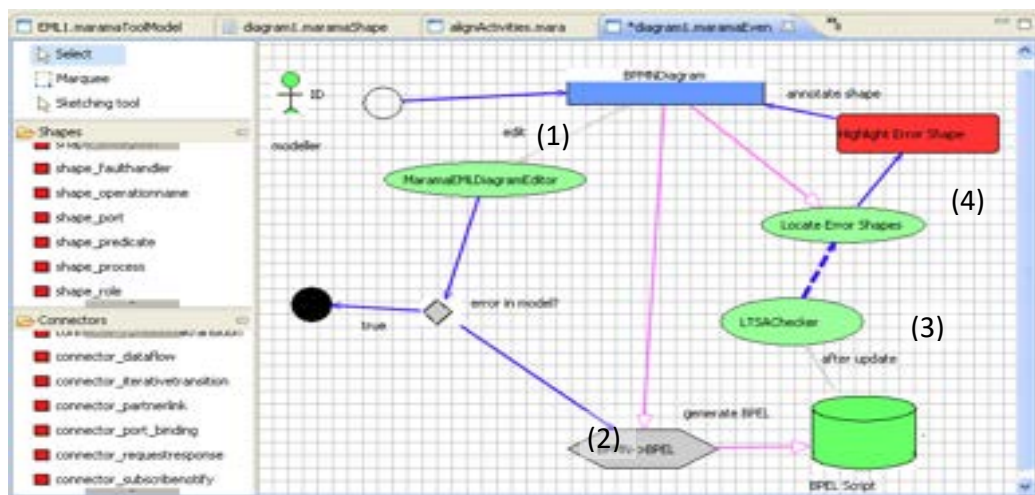


Figure 12. Event-based tool architecture integration specification.

Figure 12 shows specification of user-defined events and propagation of their notifications between various MaramaEML event handling toolies and structural components. This example specifies that when a BPMN diagram is modified and has no constraint violations (1), a BPEL script is generated from the underlying BPMN meta-model entities and associations (2). This BPEL script is then submitted to an LTSA-based model checker (3). This performs a number of consistency checks on the model. Errors detected by the model checking are used to open, annotate and highlight in the BPMN view (4).

5. 2. 6. Integrated runtime visualisation support

Visualisation support for a running DSVL tool is also necessary to allow users to track and control system behaviour using the same level of abstraction as they are defined in [26, 28]. Synthesised runtime visualisation is achieved in Marama via a specialised debugging and inspection tool (a “Visual Debugger”). Our Marama Visual Debugger provides a common user interface that connects the three metaphoric event specification views with an underlying debug model based on the model-view-controller pattern. We use the debugging service instrumentation mechanism [53] to generate low-level tracing events on modelling elements. Marama handles those events by sending the event data to appropriate modelling elements and annotates them with colours and state information. Marama EMF is the common high-level representation that glues different behavioural views together, and supplies dynamic state information to the Visual Debugger. The user has full control of execution, with step-by-step visualisation of results (e.g. query results or state changes) at the point of execution of each building block in a particular view. Figure 13 illustrates the visualisation of an event handler (a) followed by a runtime-interpreted formula (b). The Meta-model Designer view and the Event Handler Definer view with the respective formula and event handler specifications are juxtaposed with the runtime diagram modelling view. From the Visual Debugger, the user has control over the execution/interpretation of a behavioural building block. Once the behaviour is interpreted, the affected runtime model element is annotated (with a yellow background) to indicate the application of the formula/handler, and meanwhile, the corresponding formula/handler node and their dependency links defined in the corresponding meta-model view are annotated in the same manner to show the behaviour specification and its execution status.

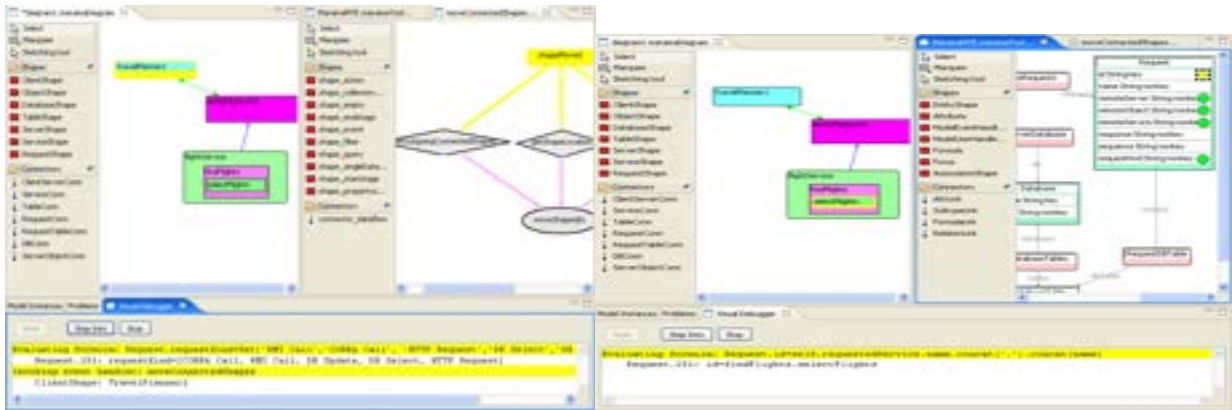


Figure 13. Visual debugging of a Kaitiaki event handler (left) followed by a MaramaTatau formula (right).

6. Critics and Transformations (Requirement 3)

Using the Marama capabilities described in the previous section, a tool developer can specify and realize a wide range of DSVL tools. However, our initial Tool Developer evaluations of Marama found two aspects that were problematic: specifying constraints and proactive design advice; and specifying model transformations and code generation. To address these we developed two further visual specification meta-tools for Marama: a Design Critic specification tool; and a Model Transformation specification tool.

6. 1. Specifying Design Critics

Research has shown that DVSL tools can greatly benefit from the addition of proactive “design critics” [8]. These critics monitor the state of DSVL tool models and provide proactive feedback to the tool user around model quality. Some critics support “fix up” actions to modify an incorrect or inefficient design structure under user direction. While these can be specified with our declarative and imperative visual event handler specification tools described in Section 5, evaluations of Marama found these to be sub-optimal approaches for most Tool Developers.

To this end we have developed a visual and form-based critic specification meta-tool for Marama, MaramaCritic Designer. MaramaCritic provides a high-level, declarative specification approach to add design critics to DSVL tools. It uses a combination of a high-level visual critic model and a more detailed form-based critic specification. Together these provide a critic definition approach that is more accessible, though more restricted in its expressability, than the other behavioural definition approaches.

The main underlying idea in MaramaCritic is to use information expressed in a meta-diagram (i.e. the Marama meta-model diagram) as input for critics to be realized in a diagram (i.e. a Marama diagram in the realized modeling tool specified by the meta-model). It is important to mention that MaramaCritic is only minimally dependent on the notation used

in the meta-diagram. As we discussed earlier, the Marama meta-model diagram is expressed using an Extended Entity Relationship (EER) notation. If a richer notation is used in the future, more information can be extracted from the meta-model diagram and, thus, can be used for specifying critics and checking user diagrams. Figure 14 (1) shows a simplified meta-model for MaramaEML comprising some of the relevant entities, attributes and associations. As shown in Figure 14(a), MaramaEML's main features include service, operation and process entities. A service entity implies a task within a business process of an organization. An operation entity represents an atomic activity that is included in a service. A process entity has two specialisations: process start and process end entities, representing respectively the start and end of a process. Associations between the required entities support the modelling of the business process structure. All services, operations and processes are organized in a tree structure to model a business process system. Figure 14 (2) shows several possible critics for the MaramaEML tool. These specify named design critics to be invoked when various events are generated by Marama model editing operations. Some critics simply provide a "critique" to the user, suggesting problems or issues with the DSVL model state. Others provide "fix up" actions to proactively correct the DSVL model or to enforce constraints on the model expressed in the DSVL. Figure 14 (3) shows a simple example of a MaramaEML structure model for a basic university enrolment service (modified from [51]). Here, the student, university, and StudyLink services are sub-services of the university enrolment service. These are represented as oval shapes. Each service may (or may not) include a sub-service. The university service includes four embedded services (i.e. enrolment office, finance office, credit check and department). Each service must include at least one operation. The operation entity is represented using a rectangle shape. For instance, the Student Service manages four operations: search courses, apply enrolment, apply loan and make payment.

The bottom-most critic in Figure 14 (2) is an example of an action assertion critic. Suppose the tool developer wants to specify a critic that constrains the service entity (i.e. *EMLService*) to have no more than four operations (i.e. *EMLOperation*). This might be sensible in order to encourage designers to split large hierarchies of services into smaller, more manageable and understandable groups as our evaluation of MaramaEML found that service entities with large numbers of operations look cumbersome to the Tool Developers. A critic can be specified for this by defining the relevant properties for event, condition and action via an action assertion template as shown in Figure 14 (4)-the form-based interface. Here, the event triggering the critic is the creation of an association link, the condition is that the cardinality is greater than 4 and the action is to delete the new association. MaramaCritic generates, from its visual and form-based specifications, a set of OCL constraints and Java event handlers that augment the generated Marama DSVL tool. When the user runs the tool, these event handlers and constraints are triggered at appropriate times by editing events. A critique (message to the DSVL tool user) is displayed if an event occurs and the model state matches that specified in a critic.

In the arity constraint example, a critique is displayed to warn the user, followed by execution of the action, as shown in Figure 14 (5). Three other critics are specified in Figure 14 (2). The first and second critics at the top specify name uniqueness constraints. A logical operator, OR is used to link the two critics so that both critics share a common feedback mechanism. The second-to-bottom critic shows a situation where one critic is dependent on another. The dependency of critics can be represented visually by using the CriticDependencyLink connector, which implies a sequence of critic execution between two critics. A critic that depends on another critic will only run when the critic it depends on is not violated. For instance, in Figure 14 (2) it shows the critic: "*EMLService name must have a unique name*" is dependent on a critic: "*EMLService name must not equal null*". This means the unique name critic is executed only if the service name is not null.

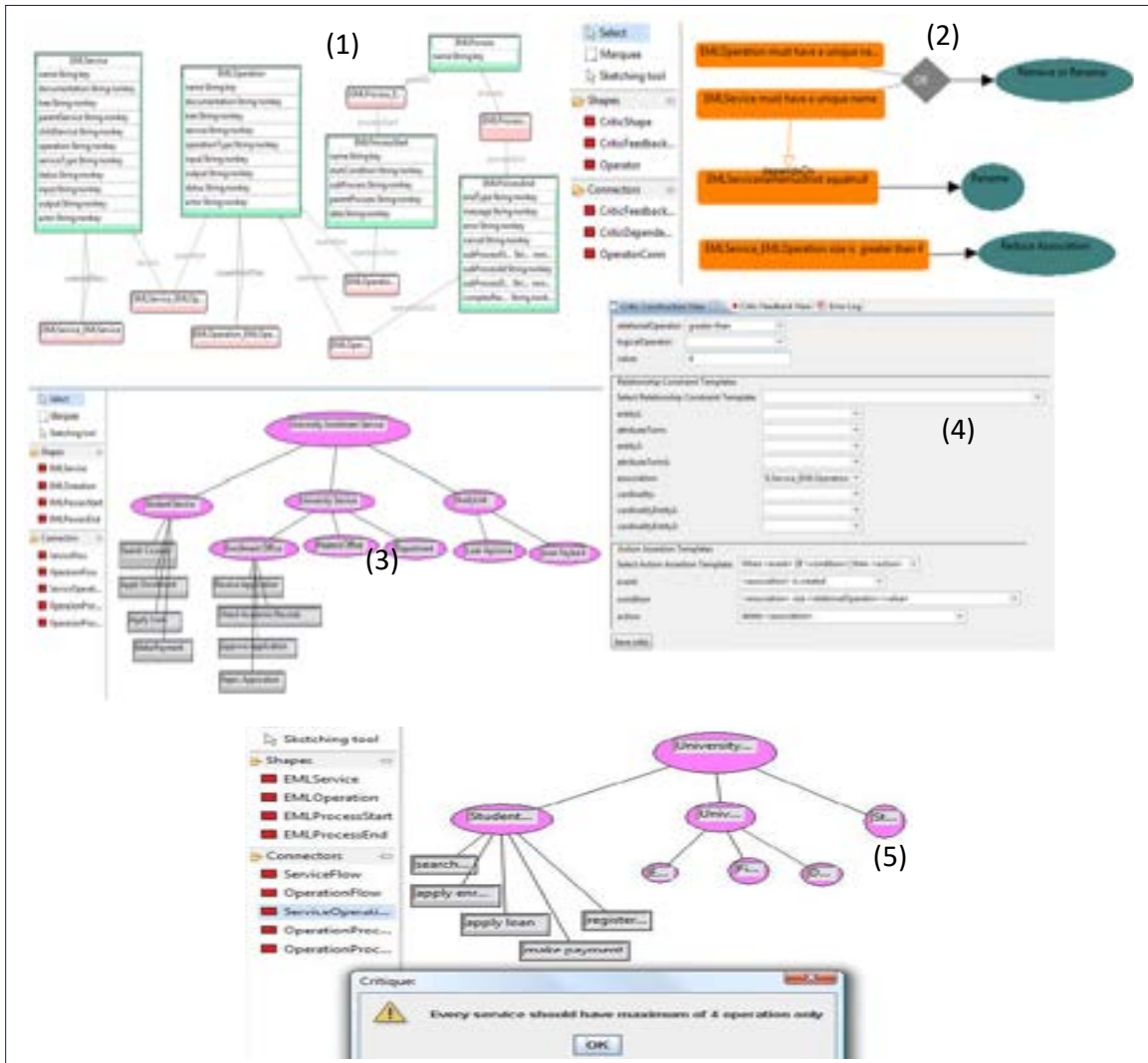


Figure 14. Meta-model for a simplified MaramaEML (1) followed by a visual and form-based critic specifications (2), an example of MaramaEML structure (3) an action assertion template (4) and critic execution (5).

These critics could all have been implemented using Java event handlers to implement similar constraint testing and feedback to the user. However, specifying constraints and feedback using event handlers is time-consuming and difficult to maintain as the meta-model evolves over time. Also, as MaramaEML has several integrated modelling notations and a canonical meta-model, it was a complex task to implement inter-notation constraints. Applying our critic designer to the canonical meta-model is straightforward; implementations of critics that took several hours to specify, test and evolve using event handlers can be done in a matter of minutes. Understanding the critics is far easier than browsing and understanding the previous individual Java event handlers, which comprised hundreds of lines of Java code with Marama API calls. In contrast, the form-based critic specifications are very clear, concise, understandable, reusable and maintainable. The trade-off, of course, is that the more accessible notation provides more restricted expressability.

6.2. Model transformation

DSVL models often need to be transformed. Sometimes transformation is from one model to another e.g. in MaramaMTE+, we need to transform a Business Process Modelling Notation (BPMN) process flow specification into parts of an architecture specification [25]. We also often need to import information from another format into a tool, such as importing a BPEL specification and transforming it into a BPMN model. To support model-driven development we often need to generate code, such as Java and C# in MTE+, or scripts, such as JMeter or BPEL. Implementing such transformations using conventional programming languages, or even with higher-level transformation DSLs like XSLT, QVT or

ATL, means that users need considerable programming expertise. Originally Marama tool developers had to write Java code in event handlers to implement code generation and model transforms. We then used a combination of XSLT transforms or ATL transform scripting languages to implement complex code generation and model transformation. Marama tool developer evaluations showed these approaches were very time-consuming, error-prone, complex and difficult to maintain.

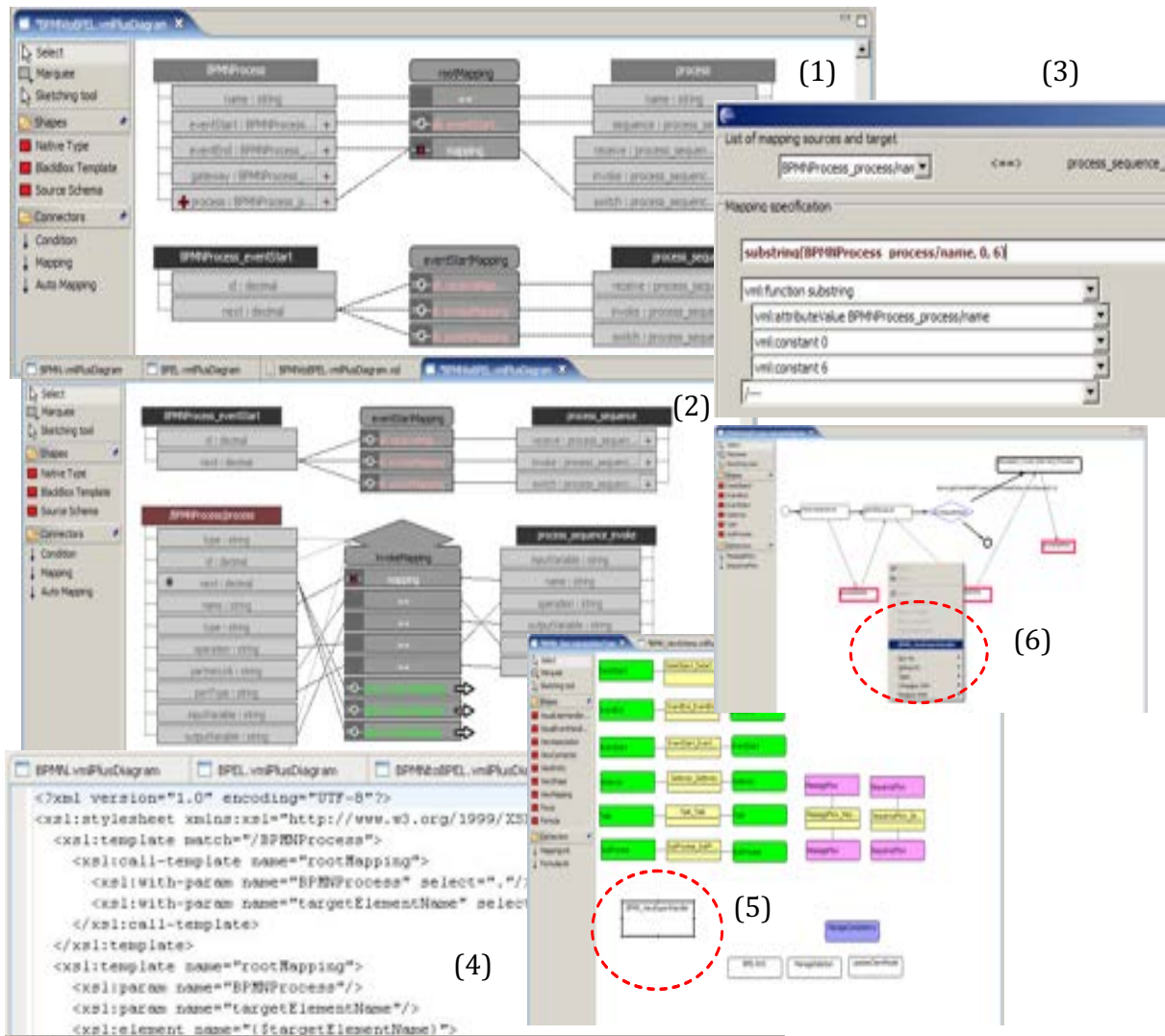


Figure 15. Model transformation specification.

To facilitate much more accessible ways to specify model transformation, model import, and code and script generation, we have developed a visual model transformation approach called MaramaTorua [37], now incorporated into Marama as the Transformation Designer. Figure 15 (1) shows an example of MaramaTorua being used to transform a BPMN notation model (left hand side) into a BPEL4WS executable representation (right hand side). In the middle is a set of transforms that map source BPMN model elements and relationships into target BPEL4WS elements and relationships. Transforms can be packaged and reused (Figure 15 (2)). Detailed formulae, including element selection, data reformatting, and iteration constructs are specified using forms (Figure 15 (3)). The visual transform is specified at the type level and is used to generate a transformer implementation. In this example, we generate an XSLT script to implement the specified model transform (Figure 15 (4)). Integrating the generated model transformation into a Marama DSVL tool is achieved by adding a view event handler to a view specification in the relevant View Designer (Figure 15 (5)). The user of the target DSVL tool then selects the transform in a diagram of this view type (Figure 15 (6)) to execute it. MaramaTorua also provides a visual debugger allowing specified transforms to be stepped through as they are run.

MaramaTorua supports XSLT and Java code generation from its visual model transformation specification models. The visual language and formulae allow quite complex multi-element and attribute model transformations to be generated. It does however have limitations when complex transforms need complex algorithmic computation, low-level data parsing

or complex iterative transformations. In this case, MaramaTorua allows Java code to be specified for the transform using an API. Such code is weaved into the generated XSLT (as Java function call) or Java code, in much the same way Marama model and view event handler code is generated and added to Marama-specified tools.

7. Human-Centric Tool Interaction (Requirement 4)

So far, we have described a set of abstract, visual specification approaches we have developed for Marama meta-tools to enable high-level specification of DSVL tools. These specifications are used to generate an Eclipse-based implementation of the target tool. However, as identified by Sutcliffe’s design metadomain approach, the generated DSVL tools ideally require a range of facilities to support what we term “human centric” modelling capabilities. These include support for collaborative modelling, sketch-based modelling and web-based modelling. We have developed a set of plug-in extensions to Marama to support each of these human-centric approaches in Marama-generated DSVL tools. We outline these capabilities in this section to demonstrate the range of human-centric modelling that can be provided for generated DSVL tools. However, while we address some of Sutcliffe’s design metadomain support characteristics, most of these should be viewed as preliminary prototypes requiring further research and experimentation.

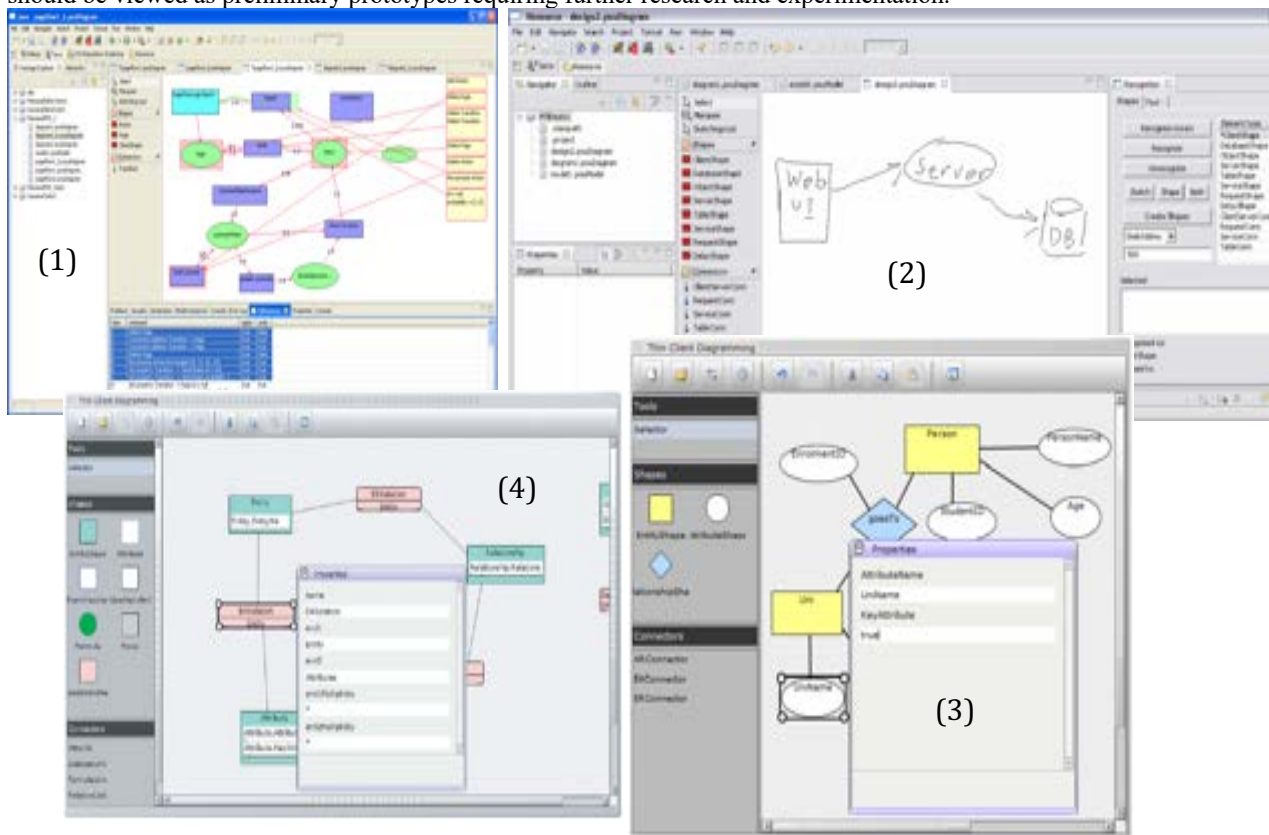


Figure 16. (1) collaborating editing; (2) sketch-based input; (3, 4) web-based editing and tool specification.

To support collaborative use of a Marama DSVL tool, we have developed a distributed support mechanism. This provides asynchronous editing support via visual differencing (MaramaDiffer) and synchronous editing support via event propagation between Marama instances [29, 61]. Figure 16 (1) shows an example of the asynchronous editing support in use. A MaramaMTE+ form chart diagram has been edited by user “john”, and user “akhil” is comparing his version of the diagram to john’s. Visual annotations are added to the Marama form chart diagram on the right hand side showing differences. A change log is shown in the view at the bottom of the screen. Akhil can select to accept all, some or no changes of john’s, and have these applied to his version of the diagram. As some edits are inter-dependent, and some are mutually exclusive, MaramaDiffer provides the user support to merge changes that are consistent. Semantic errors are highlighted in the Eclipse Problems view by appropriate design critics specified for the DSVL tool.

Sketch-based interaction with design tools is an interesting alternative approach to conventional drag-and-drop diagramming tool interfaces [16, 68]. We wanted to support this approach for Marama design tools as early-phase design has been shown to benefit from this less rigid interaction approach. We developed a new plug-in, MaramaSketch [24]

that provides an overlay for Marama diagrams allowing sketching-based input and manipulation of diagram content along with associated shape and text recognition support. Figure 16 (2) shows an example of a user drawing content (in this example with a Tablet PC stylus) onto a MaramaMTE ArchitectureView diagram. The user simply selects the sketching tool (highlighted in the left hand side editing palette) and draws with the mouse/stylus on the diagram canvas. In this example the user has drawn a *ClientShape* (rectangle, “Web UI”), an *ApplicationServerShape* (oval, “Server”), a *DatabaseShape* (cylinder, “DB”) and two connections between shapes. As each set of strokes is completed MaramaSketch recognises the shape type and remembers this.

The Marama DSVL tool diagrams shown so far require use of a desktop Eclipse IDE. Rich internet applications (RIAs) do not require desktop application installation but instead provide web browser-based access to information. To make Marama DSVL tools more widely accessible we have developed a RIA diagramming component, MaramaThin. This is implemented by provision of a set of services within Marama that allow a remote client to query diagram specifications (diagram meta-descriptions) and state (diagram model data) via XML messages. The remote clients can also update the state of these diagram models i.e. modify the diagrams. Figure 16 (3) shows an example of our browser-based diagramming tool MaramaThin in use. User “john” wants to do some ER diagramming for a MaramaMTE+ database. John opens an existing MaramaMTE+ database model diagram. John then begins to edit the diagram. The MaramaThin user interface provides a tool palette, basic editing command toolbar, an ER diagram made up of shapes and connectors that can be manipulated via drag and drop in browser, and pop-up property windows. In this example, John is editing the properties of an Entity “UniName”. MaramaThin also provides web-based access to most Marama meta-tools, including the meta-model editor, shape designer, view type designer, and constraint and event handler designers. This allows Marama/Thin users to modify their diagram specifications, or even create whole new diagramming tools, using their web browser instead of the Marama Eclipse desktop client. In Figure 16 (4) John is viewing the meta-model for the MaramaMTE+ ER diagrams. He may add new constraints to the tool using the Formula meta-model shape, modify existing constraints, or add a new property, element or association.

8. Architecture and Implementation (Requirement 5)

We have realised Marama as a set of Eclipse IDE-based plug-ins using a range of third party Eclipse projects and plug-ins. This was major departure from our earlier Pounamu meta-tool [79], where we built the whole tool infrastructure. Leveraging the range of Eclipse tool-building projects, makes it easier to integrate Marama-generated DSVL tools with other tools, and allows other Eclipse community members to use Marama in their own Eclipse-based tool research.

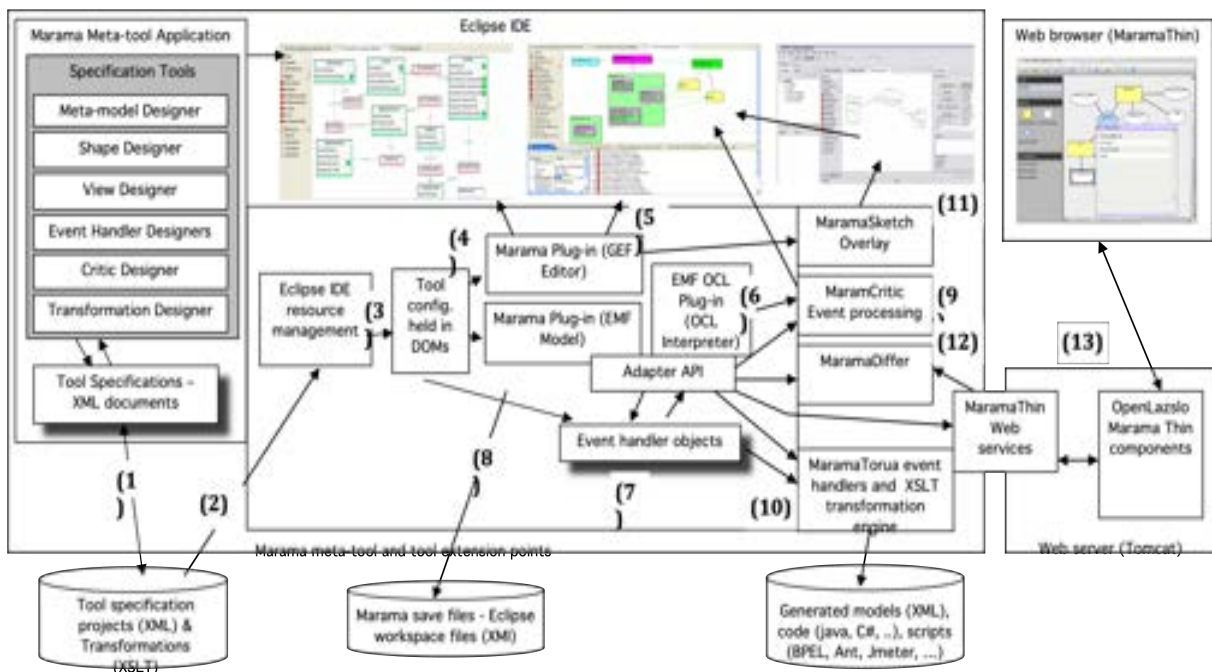


Figure 17. The basic architecture of Marama.

Figure 17 shows the architecture of the Marama meta-tools and Marama-generated DSVL tools. Marama provides a range of visual specification meta-tools. Each Marama meta-tool itself provides an editable domain-specific visual language used to specify aspects of a target DSVL tool. Tool developers initially specify a description, comprising evolvable instances of the meta-DSVL models, of their desired DSVL tool using the various visual specification tools in Marama. This DSVL tool specification is opened by a tool user and used to construct an instance of the specified tool. The tool user can then create multiple DSVL models based on this meta-description. Tool developers, and even tool users, can use the Marama meta-tools to modify a tool specification, often when the tool is “live” (i.e. in use). There are limits to this e.g. if meta-model descriptions are significantly modified then an old tool model instance may need to be transformed to the new format before being edited.

Tool specifications are stored as XML documents in a tool specification repository as shown in Figure 17 (1). DSVL tool users locate a desired existing DSVL tool specification to open or request one to be created via the standard Eclipse resource browser (2). When a tool is opened or created in Marama, the corresponding XML tool specifications are read and loaded into XML DOM objects (3). These are parsed and provide an in-memory representation of the Marama tool configuration, which is used to configure an Eclipse Modelling Framework (EMF)-based in-memory model of both the DSVL tool model and view data i.e. the properties of all entities, associations, shapes and connectors. It is also used to produce the editing controls of Eclipse Graphical Editing Framework (GEF)-based diagram editors i.e. the allowable shapes and connectors, their rendering and properties editing (4). When a diagram is opened, Marama configures a GEF editor and renders the diagram (5).

An OCL interpreter is used to implement the MaramaTatau modelled constraints on tool model and view elements (6). The MaramaTatau meta-tool designer compiles the user-specified constraint formulae into OCL definitions that are stored with the tool specification XML. When a tool is opened, these OCL definitions are loaded, compiled into the Eclipse OCL API representation used by the Eclipse OCL plug-in, and a set of event handlers used to detect changes to model data structures (add/update/delete element or association). These changes are used to trigger recomputation of OCL expressions in much the same way spreadsheet recomputation occurs. Constraints and user feedback are implemented by Marama API calls triggered by MaramaTatau-inserted OCL expression function calls.

Marama provides a comprehensive API that allows access to all of the Marama data structures and modification of these data structures via Java “event handler” code calls. This Marama API also allows controlled access to the Eclipse GEF drawing APIs and Eclipse plug-in extension mechanism. This allows expert tool developers to extend and augment Marama’s core capabilities and support very complex tool integration. Generated event handler and other behavioural and transformational objects are loaded by Marama adapter and Eclipse plug-in APIs when a tool is opened, or when the event handler definitions and code are updated in one of the Marama meta-tools (7). These Java coded event handlers are triggered when Marama model or view data structures are modified or when certain events occur e.g. button or menu selections or extension point-trigger tool events. Marama uses EMF’s XMI save and load support to store and load modelling project data (8), all managed within the Eclipse resource workspace. We have developed a set of extension points for Marama allowing new meta-tools to be created and integrated into the Marama meta-tools suite. We have also created a set of extension points allowing multiple Marama and third party tools to be integrated within a target Marama DSVL tool framework. Our event-based architectural modelling and integration visual language uses the event handler API and Marama extension points to achieve this tool integration.

Expert tool users can code complex Java event handlers to achieve sophisticated tool behaviours. We also use Java code to implement event handlers that are generated from the imperative event handler specifications from Kaitiaki and MaramaCritic specifications and the declarative MaramaALM specifications. Kaitiaki and our architectural-level event handler specifications generate event handlers to encode their event-condition-query-action models. Typically tool developers use them to achieve imperative event-based behaviours difficult or OCL-based constraints impossible to code using MaramaTatau. MaramaALM augments the shape and view type definitions and also adds generated event handlers to achieve tree and force-directed layouts for specified view types.

MaramaCritic generates a combination of OCL and Java event handler implementations, augmenting the tool specifications generated by the meta-model and view designer meta-tools. It also has a set of event handlers (9) used to coordinate processing of events, provide various critiques to tool users (via dialogue boxes and Eclipse problem markers), and carry out semi-automated “fix ups” of problematic model constructs.

The MaramaTorua model transformation meta-tool generates XSLT transformation scripts, saved with Marama tool specifications, along with Java code event handlers used to invoke these transformations in target DSVL tools (10). These event handlers convert a Marama DSVL tool model to an XML representation via EMF, then use an XSLT engine to transform the EMF structure to the target model (into XML), code (e.g. Java or C#), or script (e.g. Ant, JMeter, BPEL etc). An extension mechanism similar to Marama event handlers allows model transformation specifications to include Java code, called from the XSLT scripts, to implement e.g. low-level data format parsing and very complex transformations difficult or impossible to specify in MaramaTorua’s visual language.

MaramaSketch is a plug-in providing a “sketching overlay” which can be used with any Marama-generated DSVL tool (11). Sketching operations on the overlay are recognised using the HHReco sketch recognition engine [33], which in turn uses a training set of example sketch elements. Sketched items or groups of sketched items are turned into Marama diagram shapes, lines or text, or groups of these. MaramaDiffer is a plug-in that allows comparing and visual differencing of any Marama DSVL tool diagrams (12). MaramaThin extracts a Marama diagram data structure into XML format using EMF object serialisation. It then transforms this into an OpenLazslo-based [7] web-diagramming component, realized using Flash in a web browser to provide a rich internet client interface (13). MaramaThin uses a set of Java-implemented web services hosted in a web server (Tomcat) to communicate between the OpenLazslo-implemented diagramming client and the Eclipse-hosted Marama tool instance. MaramaThin uses MaramaDiffer to update the Marama tool diagrams, allowing multiple users to collaborate on diagram editing via a web infrastructure.

MaramaDiffer takes two diagram versions and applies a differencing algorithm to their EMF-based data structures. It determines a set of changes that would transform one diagram into the other and visually presents these as diagram annotations and a list of differences [61]. It provides support for multiple diagram version management including branching and merging. While this supports diagram merging, it does not fully support model version management, which has limited current support in Marama tools. As tools evolve, their models need to be updated along with their event handlers. Currently our Marama prototype provides limited, though not complete, support for this. Basic model version updates can be incorporated including new entities, relationships and limited change (renaming) of existing entities and attributes. However, transformation scripts need to be written to transform an old model to a new one for more complex changes e.g. split/merge entity, move attributes between entities, etc. The MaramaTorua meta-tool can be used to aid this. Marama event handlers are saved as Java classes and can be versioned using conventional repository check-in/check-out/revision processes. Marama supports simple management of meta-model version and event handler code version. Marama meta-tool models can each also be versioned, though limited support for configuration management is currently provided.

9. Evaluation

It is not a straightforward task to evaluate a substantial environment/toolset such as Marama, as it involves multiple points of views including those of meta-tool developer, Tool Developers of developed tools, usability and utility and an enormous number of variables [79]. Most formal usability evaluation approaches are limited to understanding the effect of one or two variables [18, 32]. Controlling for variability is thus an impossible undertaking when assessing the usability of a large environment. We have therefore adopted a variety of less formal, but overlapping approaches to obtain a range of evidence for usability and efficacy. Firstly, we, and our industrial collaborators, have used Marama to construct a range of different modeling tools, some very sophisticated. These provide a proof of concept demonstration that Marama is fit for purpose: it can be used to specify and realize a wide variety of DSVL environments. Secondly, we have undertaken a variety of formal and informal Tool Developer evaluations of both the core aspects of the Marama environment, and individual component extensions to it. Combined these evaluations demonstrate both efficacy and Tool Developer acceptability of our Marama approach.

9.1. Experience

We have used Marama to construct a variety of modelling tools in addition to MaramaMTE+, the exemplar used throughout this paper. The Marama meta-tools themselves (Meta-model Designer, Shape Designer, View Designer, Event Handler Designer, Critic Designer and Transformation Designer) are all implemented using Marama in a bootstrapped manner and were the first substantial exemplars developed. Four other major model-driven DSVL-based development tools are illustrated in Figure 18.

MaramaMTE+ [15] is a performance-engineering tool providing two key domain-specific visual languages and a code and deployment script generator. An architectural DSVL is used to model multi-tier architectures for systems, as illustrated on a simple example in Figure 18 (1). A stochastic form chart DSVL is used to model probabilistic loadings on the defined application. From these models, MaramaMTE+ generates Java or C# code, JMeter or Microsoft ATC load testing scripts, Ant compilation and deployment scripts, database schema and definitions, and various other deployment descriptors for J2EE and .NET applications. It then uses remote agents to run the generated load tests on this model generated software system, captures the performance results, and visualises the results in the architecture DSVL diagrams. MaramaMTE+ was part of a PhD project and took 2-3 weeks to specify and generate the core modelling capabilities using an early version of Marama. The code generators and visualisation support took the bulk of the remaining several months work, including experimentation on numerous architectures, target platforms and load models.

MaramaSUDDEn, Figure 18 (2), provides an environment for integrated supply chain modelling [35]. This allows Tool Developers, supply chain managers and SME owners, to model and configure supply chains. It includes support for reuse of supply chain models, complex diagram layout control, and integration with external tools to store, validate and

enact supply chain models. MaramaSUDDEEN was developed iteratively over a few weeks elapsed as part of a collaborative project. Much of this elapsed time was spent waiting on globally distributed partners to provide feedback on each iteration.

MaramaDPML (Design Pattern Modelling Language) provides a tool to model and instantiate Design Patterns in a UML-based design tool [58]. An example design pattern instantiation diagram is shown in Figure 18 (3), showing the Abstract Factory pattern being instantiated in a UML design for a GUI library project. MaramaDPML provides code generation from the UML models to Java code, multiple, integrated design diagrams with inter-diagram consistency management, and validation of design pattern usage in UML designs. Tool Developers are software designers. MaramaDPML was developed over a 4-5 week period based on an earlier prototype using the JViews DSVL tool framework. The main modelling framework took only a few days to specify; the bulk of the implementation time was for back end code generation etc.

The final example, Figure 18 (4), is MaramaVCPML (Visual Care Plan Modelling Language) [43]. MaramaVCPML is used to model generic care plans for chronic disease management, which are then instantiated for individual patients. Care plans include modelling of treatments, physical exercise programmes, food, and review of key health indicators. Template generic care plan models are reused for multiple patients. MaramaVCPML generates a full mobile device application implementation allowing patients and their physician to monitor patient progress against their individual care plan. MaramaVCPML supports multiple model integration, code generation and model template reuse. MaramaVCPML was developed as part of a Masters project over a 3-4 month period.

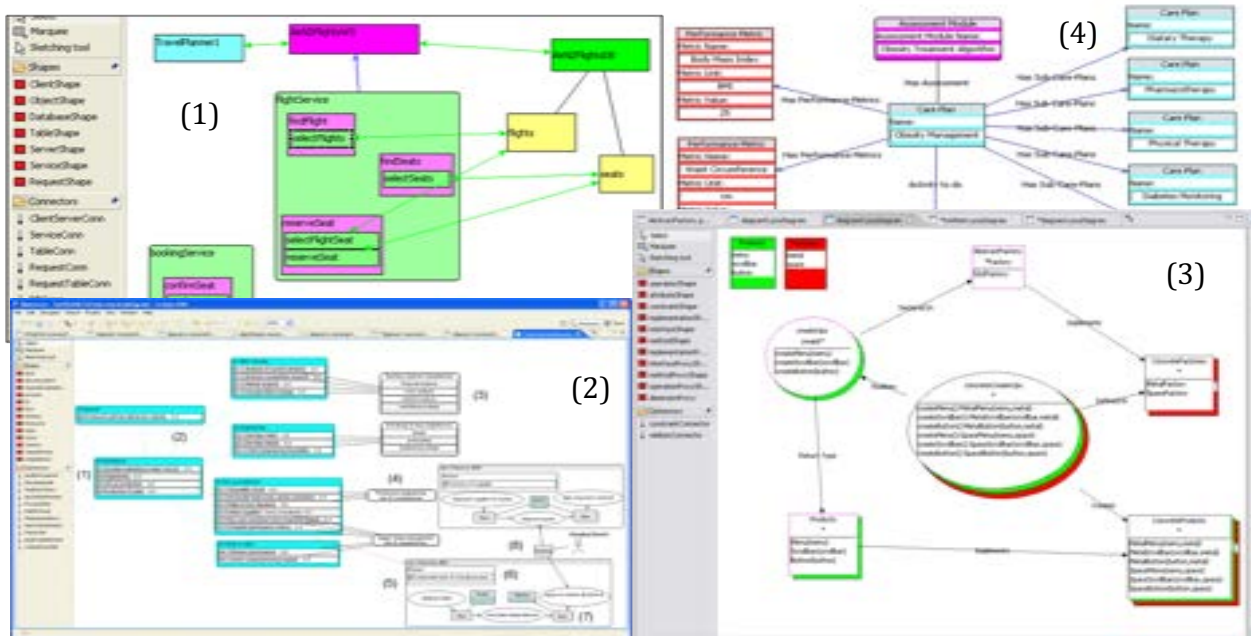


Figure 18. Examples of Marama generated tools: (1) MaramaMTE+; (2) MaramaSUDDEEN; (3) MaramaDPML and (4) MaramaVCPML.

9. 2. Formal Evaluations

We have undertaken a significant number of qualitative and quantitative evaluations of DSVL tools developed using Marama [35, 51, 58]. These have been uniformly positive in their overall appraisals of the developed tools. Feedback from tool developers using Marama to produce these tools has also been very positive. We have used Marama over several years in advanced visual language courses, including setting tool development tasks using Marama and surveying the post-graduate students for feedback on core Marama functionality. We have carried out focused qualitative and quantitative studies of individual Marama components to test their usability and effectiveness for specifying DSVL tools and to identify potential problems. We, and a small number of others, have used Marama on several industrial DSVL tool development problems. These evaluation results have been sufficiently positive for us to release Marama as a publicly accessible toolset. We summarise this range of evaluations in the subsections below.

9.2.1. Cognitive Dimensions Evaluations of Individual Marama Components

We have conducted several Cognitive Dimensions (CD) [23] evaluations as we developed Marama capabilities to inform the design of the visual meta-DSVL specifications. A CD evaluation provides an understanding of usability tradeoffs and hence where mitigations need to be placed without a need for heavyweight conventional usability evaluations. Typically, as we have developed modeling extensions, such as Kaitiaki and MaramaCritic, we have undertaken an individual CD assessment of that extension. These have taken two forms: 1) applying CDs ourselves directly in a similar manner to that proposed by [23] to explore design tradeoffs and 2) using Tool Developer evaluations coupled with a CD-based survey instrument adapted from [10], which reframes CDs into questions understandable by Tool Developers but which allow a CD analysis of their perceptions to be undertaken.

Direct CD Evaluations

To illustrate the former approach, consider the Kaitiaki visual handler specification component. In CD terms, we can describe the tradeoffs made as follows (where CD dimension names are in italics). As our target users are inexperienced programmers, we have chosen a low-to-medium *abstraction gradient* based on iconic constructs and data flow between them. The abstractions support query/action composition allowing users to specify Marama data and state changing actions as discrete, linked building blocks. The abstractions require *hard mental operations* but are mitigated by concrete Tool Developer domain objects. We have experimented with elision techniques to allow concrete icons and abstract event handler elements to be collapsed into a single meaningful icon. The dataflow metaphor used to compose event specification building blocks seems to map well onto users' cognitive perception of the metaphor, providing good *closeness of mapping*. The current approach has reasonable *visibility* and *juxtaposability*. Information for each element of an event handler is readily accessible. The event handler specification can be *juxtaposed* with the modelling view that triggers its execution. However, it still has the usual "box and connector" diagram *viscosity* problems; the user typically has to rearrange the diagram to insert elements.

As another example of this approach, we undertook a similar evaluation for MaramaTatau, the OCL constraint specification designer. In developing MaramaTatau, our focus was on providing a compact and accessible constraint representation for Marama, while minimising *hidden dependency*, *juxtaposability* and *visibility* issues. The visual abstractions introduced are visual iconic constructs and data dependency links between them. This is quite a *terse* (low *diffuseness*) extension to the existing metamodel notation and the abstractions are quite low level, providing a simple overview of constraints and dependencies, and hence have low *abstraction gradient*. Error proneness has been reduced significantly. The pre-existing Marama Java-based Marama event handler designer is very error-prone for both novice and experienced users due to its reliance on API knowledge and Java coding together with the numerous hidden dependencies with the visual metamodel. MaramaTatau reduces *error proneness* by avoiding API details and directly using concepts visible in the metamodel. The *verbosity* (high *diffuseness*) of the textual OCL, due to its many built in functions, does, however, present similar opportunities for error as does API mastery. The *verbosity* also introduces some degree of *hard mental operations* as users must remember what function is appropriate for a given purpose. However, the relative familiarity of OCL with the target Tool Developer group mitigates this and also means good *closeness of mapping* for them. The compact nature of the representation, point and click construction, and automatic construction of the visual model annotations, means *viscosity* is low. MaramaTatau allows *progressive evaluation* of a constraint specification via Marama's live update mechanism. Modifications to formulae take effect immediately after re-registration in a Tool Developer tool. A visual debugger allows users to step through a formula's interpretation using the same abstraction level as they were developed in. By contrast, java event handlers require conventional java debuggers and a good knowledge of Marama's internal structure.

Similar evaluations were undertaken for MaramaSketch and MaramaALM [27, 66], where the emphasis is more on environmental factors than just on the notation. For MaramaSketch, much emphasis was placed on minimising *premature commitment*, through deferment of recognition of sketch elements, and support of *progressive evaluation* via the ability to recognise on demand. *Viscosity* is much lower than pen and paper sketching equivalents and *closeness of mapping* was central to our motivation i.e. that sketching is a more natural mechanism for expressing initial designs than standard computer diagramming approaches.

MaramaALM offers a terse notation with simple abstractions (*low abstraction gradient*) and low *viscosity* by encapsulating the low-level implementations into a generalised component that can be easily applied to any Marama-generated tools. It also assists in reducing *viscosity* problems in the generated modelling tools by providing automatic layout in those tools. The tool-designers can change the involved shapes and connector in one place and this modification will be reflected throughout the whole mechanism (relatively low *hidden dependencies*). However, the approach comes with the cognitive dimensions trade offs of some *hidden dependency* issues and *premature commitment* problems. During the specification process, the use of Shape Designer and View Designer is inseparable due to the structural dependency between both of them. Each depends on the other to generate necessary properties and manage shape-entity mappings in order for MaramaALM to function properly; hence from one view there is a *hidden dependency* to the other. *Premature*

commitment is required, as meta-modellers need to decide which shapes and connectors are to be included in the layout support during the specification process.

Cognitive Dimensions-based Survey Based Evaluations

As an illustration of the second CD evaluation approach, we conducted a user evaluation for our MaramaCritic tool with twelve volunteer researchers and students who had basic background knowledge of the Marama meta-tools and who were interested in modeling and the development of modeling tools to support their work. The CD-based survey tool provided questions targeted at each of the cognitive dimensions as we were interested in the tradeoffs amongst those dimensions that respondents observed.

MaramaCritic offers good *visibility* and *viscosity* for the target Tool Developers. Eleven out of the 12 respondents answered that it is easy to see various parts of the tool and make changes. The only respondent who reported otherwise commented that, due to a lack understanding of meta-tool concept and as a novice user, it was hard to understand the functionality of various parts of the tool. *Diffuseness* refers to the verbosity of language, i.e. the number of symbols required to express the meaning. Ten respondents reported the notation to be succinct and not long-winded. Participants noted that the notation is a straightforward representation of a critic and its feedback, as well as the connectors that link them. MaramaCritic suffers from some *hard mental operations* (degree of demand on cognitive resources): four respondents claimed that they needed to think carefully about the use of critic templates for specifying a critic. Four respondents disagreed and four were undecided. MaramaCritic is not regarded as being *error prone* as five respondents claimed that the notation is very straightforward and supported by form-based interfaces that are familiar to most users. The respondents that answered otherwise raised the issue that unfamiliarity with the templates can cause users to make mistakes in specifying critics. MaramaCritic provides good *closeness of mapping*. All respondents noted that the MaramaCritic editor provides a notation closely related to the domain. *Role Expressiveness* for MaramaCritic is obvious as nine respondents answered it is easy to tell what each part is for when reading the notation. Ten respondents said that the dependencies are visible and two respondents are undecided. *Hidden dependencies* are primarily between the visual critic definer view and the form based template views. Moody argues that this type of hierarchical dependency is of positive benefit in his Principal of Complexity Management [65]. MaramaCritic supports *progressive evaluation* well. Eleven respondents stated it is easy to stop and check work progress. Critics and feedbacks properties can easily be edited and any new changes will take effect during the model execution of the tool. All respondents agreed that there are no *premature commitments* in the Marama Critic Definer view. The user can freely specify a critic using any critic templates. However, the user needs to define a critic first before a critic feedback can be specified and linked with the defined critic. The user can add a critic as well as the critic feedback for the Marama tool incrementally as he/she encounters new critics.

The survey results show our respondents have a good degree of satisfaction with our critic design tool integrated with the Marama meta-tools. The survey results demonstrated that for most respondents our approach appears to be useful in assisting these respondents in the critic specification task. Our approach also appears to nicely complement the other components of the Marama meta-tools and is integrated with these. However, limitations of the tool are also revealed through the survey results. Thus, some minor improvements are needed to improve the usability of the critic specification editor integrated with Marama meta-tools. The feedback and limitations that were identified from the survey have led us to refine the current critic specification editor and develop an improved critic specification editor.

9.2.2. Large Scale Tool Developer Usage of Core Marama Functionality

We used Marama over four years in two final year undergraduate and post-graduate courses. Student Tool Developers were set a project to build a DSVL tool of their choice and use a range of “core” Marama functionality. We define the “core” Marama functionality as the meta-model designer (excluding constraints), shape designer, view designer, visual OCL editor, and basic event handler specification (non-visual). These student Tool Developers documented their work in a short report which included tool description, tool usage example, and reflection on using Marama to develop their DSVL tool. We collated these reports to analyse the range of DSVL tool domains and complexity, usage of Marama features, and feedback on the Marama prototype used. These experiences help us to understand whether student Tool Developers found the Marama approach and these key Marama meta-tool features easy and effective for realising their chosen DSVL tools. We used the Tool Developers’ feedback to improve Marama, and Marama meta-tool enhancement was undertaken after every iteration.

277 graduate-level student Tool Developers participated. These were fourth year Computer Science or Software Engineering students, i.e. novice short-term research task-oriented users: in 2007 - 121; 2008 - 59; 2009 - 77; and 2010 - 20. They had used many software tools, though few had used tools for meta-modelling similar to Marama. Nothing should be read into the variation in numbers of participants between years: this just reflects cohort availability and time available by the researchers to undertake the DSVL tool development tasks.

These student Tool Developers were asked to construct a DSVL tool of their choice, but with at least a minimal set of required components, so that tools with a realistic level of complexity were developed and participants were required to exploit a core set of the Marama functionality. The required component set included: appropriate numbers of metamodel entity types and associations; appropriate shapes, possibly of differing complexity (of the icon) and connectors; at least two different view types, i.e. showing different types of information within each view type; a few simple event handlers managing things such as diagram layout, editing constraints, model (entity) constraints, mock code generation, data import, etc. Preparatory training sessions were provided on: (i) general DSVL design concepts and principles; (ii) a general introduction to meta-tools and metamodeling concepts; and (iii) a specific introduction to Marama architecture and implementation. Student Tool Developers were also provided with exemplar tools, demonstrations and tutorials. The Tool Developers worked individually, in pairs, or in teams of 3. The larger the team the more complex the tool required.

Tool developers were then given three weeks elapsed time (working alongside other commitments) to complete the prototype development together with an individual survey report answering a set of open ended questions to qualitatively express both strengths and weaknesses of Marama in constructing their desired DSVL tool. We analysed the open-ended surveys and categorised the responses into nominal qualitative attributes representing the strengths and weaknesses of Marama from the Tool Developer's point of view.

184 DSVL tool instances were developed using Marama in the projects (in 2007 - 93; 2008 - 43; 2009 - 43; and 2010 - 5). These included, in order of popularity, data modelling (69), component and architecture design (58), process modelling or planning (23), management (22), interface design (7), and requirements modelling (5) tools, summarised in Figure 19. This indicates that Marama can be effective in application across a large breadth of DSVL application domains. Our set of basic requirements, described above, was met in 69% (127 out of 184) of the tools developed. Among those 57 tools that failed to meet all basic requirements, 1 failed the shape definitions (due to an invalid shape design not reflected in the shape viewers), 33 failed the view type specifications (either omitting a view type or containing unloadable information in the views), and 36 failed the event handler implementation requirements (either not functioning with runtime errors or not implemented at all, with reported reasons including the steep learning curve, difficulty of debugging and time constraints). Many struggled with the MaramaTatau visual OCL constraints in earlier years due to problems with its implementation.

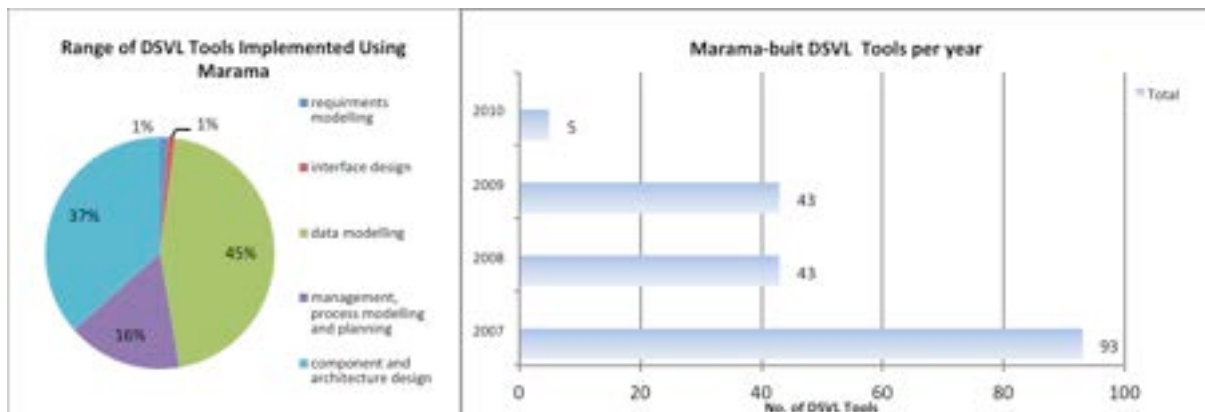


Figure 19. Broad domains of DSVL tools developed with Marama (left) and tools developed per year (right).

Most Tool Developers responded in the surveys that Marama was suited to develop their tools effectively (i.e. they were able to achieve the requirements of the assignment) and efficiently (i.e. they were able to achieve their desired DSVL tool results in a “reasonable” time frame) in general. However, there was still room for improvement. We specifically asked students for lists of both strengths and weaknesses, so a comparison of the absolute numbers of strength and weakness responses is not appropriate, however a categorisation of each is instructive.

Figure 20 (a) shows a broad categorization of strengths identified by Tool Developers in their reports. Strengths highlighted include: the rapidity of constructing DSVL tools; the simple approach for defining tool data structures with separation of concerns and a seamless integration process; and the consistent, easy to understand metamodel, visual notation and multiple view type abstractions. These reflect the core aims of Marama: the ability to efficiently construct new DSVL-based modeling environments. Lesser numbers of comments related to extensibility and customizability. Comments in reports from tool developers here include ones relating to the powerful event handling mechanism for extending tool behaviours; and the low effort needed and minimum hidden dependency issues arising when customising and constraining Tool Developer models and views effectively in the generated tools. The lower numbers here probably reflect that these features are secondary in novice tool developers' minds to those of core meta-model and view definition.

Figure 20 (b) categorises the weakness responses. The largest categories concerned the general Marama development environment, and its stability and performance. Participants compared Marama to typical robust open source software and thus expected it to have sound API documentation (access to API documentation is needed for complex event processing) and comprehensive user manuals to help smooth the initial steep learning curve. Neither of these is available to the level of quality desired, due to the research prototype nature of the system. Similarly, various modern IDE capabilities were noted to be lacking by different participants. These included automated searching and registration, automatic layout, model validation and refactoring, progress tracking, dynamic debugging, copy/paste, undo/redo, automatic backup and version control, multiple platform portability and collaboration support. Many of these had been addressed in the research branches of the toolset, but had not found their way into the core branch released to the Tool Developer participants. Stability and performance issues also irritated Tool Developers, including the need to: fix a variety of annoying bugs; provide more user friendly error messages and handle errors more gracefully; roll back unsuccessful operations; reduce memory and resource consumption; and optimise the edit-compile-run and dynamic loading processes. These weaknesses are all typical of the leading edge development of complex software environments and hence, while irritating, did not give us cause for concern over the viability of our approaches.

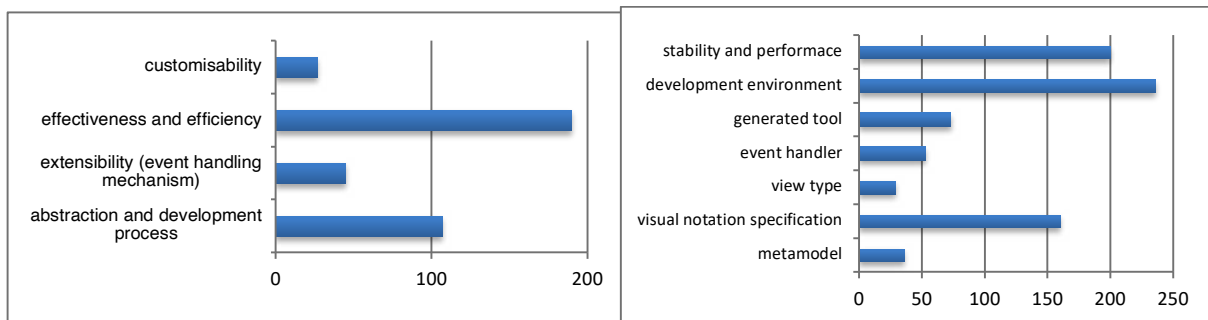


Figure 20. Strengths of Marama reported from student Tool Developer reports (left); and weaknesses reported (right).

The most significant “non-trivial” shortcoming of Marama was its limitation for visual notation design. Tool Developers found the range of shape/connector types, the flexibility of drawing mechanisms, and the configuration of visual variables (e.g. layout, texture) to be lacking. This category accounted for over half of the remaining identified weaknesses. Respondents also, but to a lesser extent, noted a variety of limitations of Marama generated tools and suggested including better modelling element identification management; allowing a “smart” connection type that dynamically infers actual types; allowing customisable toolbox items, icons and tooltips; and allowing deployment of generated tools as standalone applications. Far smaller numbers of comments were made regarding the metamodel, view type and event handler specification elements. Typical suggestions included providing an n-ary association type; allowing multiple associations between two entities; allowing association sub-typing; adding more wizard/dialog or automation support for view mapping; allowing linking of multiple view types passing common values in between (i.e. auto mapping between meta model elements); and providing more comprehensive event handler building blocks for composition and reuse. Of note is that most of the non-trivial shortcoming comments were in fact in the form of suggestions for fairly straightforward enhancements, many we incorporated into subsequent iterations, rather than fundamental issues with our approach.

We conclude from these experience reports that the Marama approach and core functionality of Marama has demonstrated itself to be a suitable platform for designing baseline DSVL notations, with excellent extension points for behaviours implemented as model/user event handlers. A key aim was to avoid having Marama users need to write complex code and use complex API calls as needed for advanced features in many DSVL toolkits and IDEs. Thus much of our research focus has been placed on using visual languages and metaphors, such as Kaitiaki, MaramaTatau, MaramaCritic and MaramaTorua, for event handling, constraint/critique authoring and model transformation specification. These research features we have evaluated for the most part individually to date.

A few participants each year used an alternative meta-tool platform to Marama. These included MS DSL Tools (a Visual Studio SDK extension), Eclipse GMF, and MetaEdit+. This was usually because they had previously used Marama on an Honors or Summer Research project. We asked them to report anecdotal evidence of their experiences using these tools compared to Marama. In general, feedback on Marama’s core capabilities was positive, with students reporting GMF and DSL tools generally required them to write much more Java or C# code respectively. They also were required to know many more details of the tool frameworks compared to Marama meta-tools in order to achieve similar DSVL tool capabilities. MetaEdit+ provided good high-level abstractions though several of these were form-based or script-based vs visual specifications as in Marama. We plan to have students use and compare other meta-tool platforms to Marama, such as VisualDiaGen and Tiger, in the future, and to more systematically compare and contrast these tools to Marama.

9. 2. 3. Other Small Scale Evaluations of Individual Marama Components

A further technique we used regularly was to use small groups of users to conduct informal, but usually insightful, evaluations. Participant numbers were typically too small for statistically significant quantitative results but sufficient for qualitative feedback. For example with the MaramaTorua mapping tool, our Tool Developer evaluation had four experienced data translator implementers carry out a set of mapping tasks (parts of the BPMN->BPEL4WS problem described earlier) [37]. They used MaramaTorua to model the schema, specify a range of mappings, generate an XSLT-based translator, and test it. Overall results were very favorable with all users able to carry out the task in orders less time than directly implementing Java or XSLT translators. Users expressed overall satisfaction with MaramaTorua's capabilities. Some difficulties found were modeling conditional mappings and string parsing operations and the specification of complex expressions using the MaramaTorua formula editor. Users desired a "design by example" approach for the latter using actual source and target values with the tool inferring conversions. Conducting such simple, fast evaluations regularly informed our development approach.

9. 2. 4. Industrial users and applications

During the development of Marama, we were fortunate to develop a strong relationship with a Model Driven Engineering consultancy company, who regularly, over the period of nearly 2 years, applied Marama to develop industrial strength visual editors that were deployed into large corporate organisations. To support this company, we developed an issue reporting and feedback mechanisms, and obtained regular qualitative feedback from the company personnel. While informal, in comparison to typical "academic" evaluations, these feedback mechanisms were incredibly useful to us. Tool developers, in this case, were professional programmers and hence differed somewhat from our originally intended target of less technically-able tool developers. As a result they preferred using the core marama meta-tool features of meta-model, shape, view definition and visual OCL, coupled with handcrafted Java event handlers (i.e. similar to those features predominantly used in our large-scale experience report above) over features more specifically targeted at less technical Tool Developers, such as Kaitiaki, MaramaCritic and MaramaTorua. We asked them for feedback on their motivation for choosing to use Marama on industrial projects, their experiences with Marama on these projects, their assessment of the Marama approach and prototype tool platform, and issues that they felt needed to be addressed in future research to improve the Marama approach for industrial adoption.

Motivation for choosing Marama was three-fold: its approach to designing DSLs, its support for modelling with a wide range of generated visual language tools, and its open-source license. Additionally, its use of Eclipse projects and close integration with Eclipse toolsets made it attractive to them. When the company explored options for a next-generation tool for knowledge engineering, they felt very few available tools had these characteristics. Marama was used by the company on two significant projects each involving development of a set of visual languages/editors and was used in demonstrators along with generated DSL tools on a wide variety of occasions. Significant development of Marama was undertaken by the company, in collaboration with our research group, to "industrially harden" the Marama prototype implementation. This allowed for more scalable and robust Marama tools to be developed and deployed, as well as making the Marama meta-tools implementations themselves more robust.

Overall the feedback provided on the core Marama features, while anecdotal, was extremely positive, with high satisfaction reported by the tool developers on the productivity afforded using Marama. Also reported was strong satisfaction by tool end-users when Marama generated tools were deployed into their client organisations. Some of the key "shining aspects" of Marama reported by the company included its approach to supporting generation of DSL editors using predominantly visual specification techniques "without confusing technical overheads" and with good separation of concerns. The ease of turning conceptual models into visual diagramming tools often "wowed" industrial partners of the company and the end users of the tools. The company judged the Marama approach was significantly ahead of other platforms such as MetaEdit+, DS Tools and GMF.

Some limitations of Marama related to its research prototype nature; the immaturity of its implementation and a range of usability limitations, some due to its reliance on Eclipse projects, and some, as also reported by our student tool developers above, due to its prototypical nature. Many potential users lacked confidence to develop Marama-based tools either on their own or exclusively relying on external parties to assist in these efforts. Much of this lack of confidence related to a lack of reusable patterns/diagrams/parts, something we have been addressing in recent work [49]. Some lack of custom graphic symbols, out-of-the-box layout managers, and reliance on Eclipse deployment approaches also drew negative comment from potential users. The greater platform and usability maturity of other DSL tools, such as MetaEdit+, was significantly attractive to some tool developers.

However, for this high-end Tool Developer, more fundamental limitations with the Marama approach, and realisation of the approach in our current prototype, were also identified. These included the lack of a shared, robust and highly scala-

ble repository for digital artefacts. This was a major requirement for many model-centric organisations. The desire for an entirely web-based user interface for both specification and generated modelling tools was also important. Additionally, a clean implementation of a denotational semantics that completely decouples modelling from naming, and a desire for a more mathematically-based meta-model including model and category theory underpinning the modelling framework was desired by our partner, rather than using less formal EER meta-models as currently used by Marama. These requirements are well beyond those of our target Tool Developer community for which a simple, readily understandable meta-model approach is preferable. Marama currently also lacks textual DSL tool integration and has no direct support for textual DSL design and generation. Many organisations adopting model-driven engineering use textual DSLs and wanted this support to be as accessible and integrated in Marama as its current support for DSVLs.

We have also used Marama ourselves on several industrial projects, predominantly developing proof of concept tools to assist in our own consulting work or to assist our industry partners in their R&D projects. Significant Marama tool projects whose results fed into industrial R&D efforts included developing an XForms designer, two business process modelling tools, a supply chain modeller, a health care plan modeller, and a requirements engineering support tool. All of these tools, developed with Marama, were used experimentally by industrial partners and their target end users. As with our partner experiences above, we found Marama to be very effective for designing new domain-specific visual languages and exploratory authoring of these by end users. The generated DSVL tools could be rapidly developed and experimented with. However, limitations with its model repository, limited support for large model visualisation, minor but annoying usability issues, and lack of textual DSL integration, all contributed to these tools not being widely deployed.

10. Discussion

Combined together, our experiences gained from complex DSVL application development, the core environment evaluations, the individual component evaluations and our industrial users indicate our Marama approach to DSVL tool engineering is effective. Our small-scale evaluations separate concerns, allowing focus on individual components, and have provided evidence for their efficacy and usability. Our more substantial whole of environment evaluation provides evidence that the components work effectively when combined together. In both cases good evidence for the efficacy and usability of Marama has been provided. Deficiencies noted, particularly from the large-scale evaluation, are predominantly related to expected software maturity issues, such as a lack of API documentation and software stability, and issues specific to non-target user groups, such as the need for a more mathematically robust meta model, rather than fundamental issues with the approach. Experience from applying Marama to realize substantial DSVL environments, both academic and industrial, allows extrapolation from the more formal, and hence restricted, evaluations, to more realistic usage. Both the developer experience in specifying these environments and the Tool Developer evaluations of their usability and efficacy provide strong support for Marama's effectiveness as a meta-tool.

While our aims in developing Marama were to afford meta-tool capability to non-technical Tool Developers to allow them to develop their own modelling environments, we are not quite at the point of generating sufficiently robust evidence to demonstrate that we achieved this. Our evaluations have focussed on more technically proficient Tool Developers to date. Conducting similar studies on non-technically proficient audiences remains future work.

We have, however, met each of the key requirements for a DSVL meta-tool we established in Section 2. Looking in more detail at Sutcliffe's Design meta-domain model [73], we see that Marama provides strong support in some areas and partial support in others, suggesting future work opportunities as follows:

- Modelling and prototyping support; visualization support

Marama's multiple metaphor meta-DSVL models provide effective separation of concerns by specifying different DSVL aspects at different abstraction levels. At a tool level, there is seamless integration of these multiple abstractions, however hard mental operations and hidden dependencies are introduced as a trade-off for the resulting specification flexibility. Users need to decide which visual language to use at a particular modelling stage. Therefore, there is a need for a description and guidelines for these metaphors, from which users can make better choices about their specification approaches. An obvious direction to proceed is to use MaramaCritic to specify a set of meta-critics for the Marama meta-tools embodying such high-level guidelines and constraints. We aim to operationalise Moody's Physics of Notations [65] design principles in these critic-based guidelines and to provide some basic visual language assessment support for evolving DSVLs within Marama itself.

In addition, the expressability provided by some of the Marama meta-tool components is currently limited, making it difficult to design and realize some types of diagram. Specifically, modeling tools are limited to box and connector, with some limited containment support. Such restrictions are common in current meta-tools. We are currently looking to extend this to allow for notations such as Euler graphs [36] which combine box and connector and overlapping region requirements.

To provide better and more integrated support for information presentation and visualisation, we aim to merge modelling with visualisation, to empower visualisation with the capability of extracting/generalising models for reuse. From this, instead of creating new models from scratch each time, we will allow users to explore existing models and capture reusable components by various visualisation functionalities such as querying, filtering and abstracting. This extends our Kaitiaki work with more complex reusable query and visualization support [54].

- Critics and knowledge retrieval/reuse

MaramaCritic provides good support for specification and realization of tool level critics. These, however, need to be manually specified by tool developers, whereas Sutcliffe envisaged some form of automated reasoning over prior solutions. Similarly, Marama currently has very limited support for reuse of designs and part designs. A promising direction we are exploring is meta-pattern support, which provides facilities for representing and instantiating domain-specific and domain-independent meta-fragments [40, 49] including a set of generalised/generic tasks (e.g. forming trees, juxtaposing multiple view display). This will provide one step towards a more intelligent approach to design reuse.

- Annotation, collaboration and creativity support

While Marama has some useful annotation, collaboration and creativity support via its collaborative editing, and web and sketch based diagramming support, there is room for enhancement. We plan to employ program-by-demonstration techniques to provide ways of recording, simulating and validating designs. Such techniques should allow users to play pre-recorded macros to learn the visual languages and their modelling procedures, and to specify their own domain systems following demonstrated examples or patterns. In addition to the current procedure of generating DSVL environments from meta-level structural and behavioural specifications, we wish to also allow users to demonstrate the intent of their DSVL tools and automatically reverse generate the specifications (both structural and behavioural) from that, with further refinement allowed via round-trip engineering.

11. Conclusions

Models are used in a huge range of domains. This provides an obvious driving force for good tools to author, visualise, manage, and evolve models. We have described Marama, a meta-DSVL tool for multi-view DSVL tool generation. The core of Marama comprises a set of visual meta-DSVL models for specifying both the structural and behavioural aspects of DSVL environments. Extensions include design critics, model transformation, collaborative editing, thin-client and sketch-based editing interfaces. We have developed a number of complex DSVL tool applications in various domains using Marama. These include performance engineering, enterprise/business process modelling, design patterns, health care planning, data modeling, software engineering, and so on. We have carried out a variety of evaluations of Marama itself and Marama-developed DSVL tools. Overall, while there are a number of areas for further enhancement of the meta-toolset, it provides an effective set of DSVL-based meta-tools for designing and realizing a wide variety of DSVL tools. Key enhancements proposed include using design critics to provide “meta-critics” and constraints for DSVL language design; support for knowledge reuse via DSVL patterns and refactoring support; and further extension and enhancement of annotation, collaboration and human-centric modeling interfaces. The Marama toolset has now been released in an open source form (<https://wiki.auckland.ac.nz/display/csidst/Welcome>) and is being taken up by a number of research groups and industrial partners for software tool prototyping. It is in the process of commercialising and “industry hardening” with SofismoAG.

Acknowledgments

The authors gratefully acknowledge the support of our colleagues Assoc Prof Robert Amor, Dr Beryl Plimmer, Dr Gerald Weber, Dr Rainbow Cai, and many project students and industry partners. We also acknowledge funding provided by The New Zealand Ministry of Science & Innovation’s Software Process and Product Improvement project, the Malaysian Government, the University of Auckland, and the Tertiary Education Commission’s funded BuildIT project.

References

1. Eclipse Graphical Editing Framework. Available from: www.eclipse.org/gef/.
2. Eclipse Graphical Modelling Framework. Available from: <http://www.eclipse.org/gmf/>.
3. IBM ILOG JViewsDiagrammer. Available from: <http://www-01.ibm.com/software/integration/visualization/jviews/diagrammer/>.
4. Merlin Generator. Available from: <http://sourceforge.net/projects/merlingenerator/>.
5. Microsoft Domain Specific Language Tools. Available from: <http://code.msdn.microsoft.com/vsvmsdk>.
6. NetBeans Visual Library. Available from: <http://platform.netbeans.org/graph/>.

7. OpenLaszlo. Available from: <http://www.openlaszlo.org/>.
8. Ali, N.M., et al., End-user oriented critic specification for domain-specific visual language tools, in Proceedings of the IEEE/ACM international conference on Automated software engineering. 2010, ACM: Antwerp, Belgium. p. 297-300.
9. Bentley, R., et al., Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system, in Proceedings of the fourth International WWW Conference. 1995: Boston, MA.
10. Blackwell, A.F. and Green, T.R.G. A Cognitive Dimensions Questionnaire Optimised for Users. in Twelfth Annual Meeting of the Psychology of Programming Interest Group (PPIG-12). 2000. CoriglianoCalabro, Cosenza, Italy.
11. Bottcher, S. and Grope, S., Automated data mapping for cross enterprise data integration, in 2003 International Conference on Enterprise Information Systems. 2003.
12. Brieler, F. and Minas, M., A model-based recognition engine for sketched diagrams. *J. Vis. Lang. Comput.*, 2010. 21(2): p. 81-97.
13. Buchner, J., Fehnl, T., and Kuntsmann, T., HotDoc a flexible framework for spatial composition, in the 1997 IEEE Symposium on Visual Languages. 1997, IEEE CS Press. p. 92-99.
14. Burnett, M., et al., Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 2001. 11(2): p. 155-206.
15. Cai, Y., Grundy, J., and Hosking, J., Synthesizing client load models for performance engineering via web crawling, in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. 2007, ACM: Atlanta, Georgia, USA. p. 353-362.
16. Chen, Q., Grundy, J., and Hosking, J., An e-whiteboard application to support early design-stage sketching of UML diagrams, in Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments. 2003, IEEE Computer Society. p. 219-226.
17. Cox, P.T., et al., Experiences with Visual Programming in a Specific Domain - Visual Language Challenge '96, in the 1997 IEEE Symposium on Visual Languages. 1997.
18. Dillon, A., Usability evaluation. W. Karwowski (ed.), *Encyclopedia of Human Factors and Ergonomics*, 2001.
19. Draheim, D. and Weber, G., Modeling Submit/Response Style Systems with Form Charts and Dialogue Constraints, *Lecture Notes in Computer Science*, 2003, Volume 2889/2003, 267-278.
20. Ebert, J., Süttenbach, R., and Uhe, I., Meta-CASE in practice: A CASE for KOGGE, in *Advanced Information Systems Engineering*, A. Olivé and J. Pastor, Editors. 1997, Springer Berlin / Heidelberg. p. 203-216.
21. Ehrig, K., et al., Generation of visual editors as eclipse plug-ins, in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. 2005, ACM: Long Beach, CA, USA. p. 134-143.
22. Gordon, D., et al., A technology for lightweight web-based visual applications, in Proceedings of the 2003 IEEE Conference on Human-Centric Computing. 2003, IEEE CS Press: Auckland, New Zealand. p. 28-31.
23. Green, T.R.G. and Petre, M., Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *JVLC*, 1996. 7: p. 131-174.
24. Grundy, J. and Hosking, J., Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool, in Proceedings of the 29th international conference on Software Engineering. 2007, IEEE Computer Society. p. 282-291.
25. Grundy, J., et al., Performance engineering of service compositions, in Proceedings of the 2006 international workshop on Service-oriented software engineering. 2006, ACM: Shanghai, China. p. 26-32.
26. Grundy, J.C. and Hosking, J.G., ViTABaL: A Visual Language Supporting Design by Tool Abstraction, in the 1995 IEEE Symposium on Visual Languages. 1995, IEEE CS Press: Darmsdart, Germany. p. 53-60.
27. Grundy, J.C. and Hosking, J.G., Serendipity: integrated environment support for process modelling, enactment and work coordination. *Automated Software Engineering: Special Issue on Process Technology*, 1998. 5(1): p. 27-60.
28. Grundy, J.C., Hosking, J.G., and Mugridge, W.B., Visualising Event-based Software Systems: Issues and Experiences, in *SoftVis97*. 1997: Adelaide, Australia.
29. Grundy, J.C., et al., Generating Domain-Specific Visual Language Editors from High-level Tool Specifications, in the 21st IEEE/ACM International Conference on Automated Software Engineering. 2006: Tokyo, Japan. p. 25-36.
30. Grundy, J.C., Mugridge, W.B., and Hosking, J.G., Constructing component-based software engineering environments: issues and experiences. *Journal of Information and Software Technology*, 2000. 42 (2): p. 117-128.
31. Guerra, E., Lara, J.d., and Diaz, P., Visual specification of measurements and redesigns for domain specific visual languages. *JVLC*, 2008. 19(3): p. 399-425.
32. Hartson, H.R., Andre, T.S., and Williges, R.C., Criteria for evaluating usability evaluation methods. *International Journal of Human-Computer Interaction*, 2003. 15(1): p. 145-181.
33. Heloise, H. and Newton, A.R. Sketched Symbol Recognition using Zernike Moments. in 17th International Conference on Pattern Recognition (ICPR'04) 2004.
34. Holt, R.C., Winter, A., and Schurr, A. GXL: toward a standard exchange format. in *Reverse Engineering*, 2000. Proceedings. Seventh Working Conference on. 2000.
35. Hosking, J., Mehandjiev, N., and Grundy, J., A domain specific visual language for design and coordination of supply networks, in Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing. 2008, IEEE Computer Society. p. 109-112.
36. Howse, J., Rodgers, P., and Stapleton, G., Drawing euler diagrams for information visualization, in Proceedings of the 6th international conference on Diagrammatic representation and inference. 2010, Springer-Verlag: Portland, OR, USA. p. 4-4.
37. Huh, J., et al., Integrated Data Mapping for a Software Meta-tool, in Proceedings of the 2009 Australian Software Engineering Conference. 2009, IEEE Computer Society. p. 111-120.

38. Johnson, G. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, 2nd Edition, McGraw-Hill School Education Group, 1997.
39. Kaiser, G.E., et al., WWW-based collaboration environments with distributed tool services. *World Wide Web*, 1998. 1(1): p. 3-25.
40. Kamalrudin, M., Hosking, J., and Grundy, J., Improving Requirements Quality using Essential Use Case Interaction Patterns, in 33rd International Conference on Software Engineering (ICSE 2011). 2011: Waikiki, Honolulu, Hawaii.
41. Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in Proc. of CAiSE'96. 1996, LNCS 1080.
42. Khaled, R., et al., A lightweight web-based case tool for sequence diagrams, in SIGCHI-NZ Symposium On Computer-Human Interaction. 2002: Hamilton, New Zealand.
43. Khambati, A., Grundy, J.C., Hosking, J.G. and Warren, J. Model-Driven Development of Mobile Personal Health Care Applications, Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, Sept 2008, IEEE.
44. Kim, C.H., Hosking, J.G. and Grundy, J.C. A Suite of Visual Languages for Statistical Survey Specification, IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), 2005, pp.19-26.
45. Krasner, G.E. and Pope, S.T., A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *JOOP*, 1988. 1(3): p. 26-49.
46. Lara, J. and Vangheluwe, H., ATOM3: A Tool for Multi-formalism and Meta-modelling, in Fundamental Approaches to Software Engineering, R.-D. Kutsche and H. Weber, Editors. 2002, Springer Berlin / Heidelberg. p. 174-188.
47. Ledeczki, A., et al., Composing Domain-Specific Design Environments. *Computer*, 2001: p. 44-51.
48. Levendovszky, T., et al., A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 2005. 127(1): p. 65-75.
49. Li, K., et al., Augmenting DSVL Meta-Tools with Pattern Specification, Instantiation and Reuse. VFP 2010, Electronic Communications of the EASST, 2010. 31.
50. Li, K.N.L., Hosking, J.G., and Grundy, J.C., A Generalised Event Handling Framework, in KISS Workshop, Workshop at 2009 IEEE/ACM Automated Software Engineering Conference. 2009: Auckland, New Zealand.
51. Li, L., Grundy, J.C., and Hosking, J.G., EML: A Tree Overlay-based Visual Language for Business Process Modelling, in ICEIS. 2007: Portugal.
52. Lin, Y., Gray, J., and Jouault, F., DSMDiff: a differentiation tool for domain-specific models. *Eur J InfSyst*, 2007. 16(4): p. 349-361.
53. Liu, N., Grundy, J.C., and Hosking, J.G., A visual language and environment for composing web services, in the 2005 ACM/IEEE International Conference on Automated Software Engineering. 2005, IEEE Press: Long Beach, California.
54. Liu, N., Grundy, J.C., and Hosking, J.G., A visual language and environment for specifying user interface event handling in design tools, in The Eighth Australasian User Interface Conference - AUIC 2007. 2007, CRPIT Press: Ballarat, Australia.
55. Liu, N., Hosking, J.G., and Grundy, J.C., MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism, in 2007 IEEE Symposium on Visual Languages and Human-Centric Computing. 2007: Coeur d'Alène, Idaho, USA.
56. Lyu, M. and Schoenwaelder, J., Web-CASRE: A Web-Based Tool for Software Reliability Measurement, in Proceedings of International Symposium on Software Reliability Engineering. 1998, IEEE CS Press: Paderborn, Germany.
57. Mackay, D., Noble, J., and Biddle, R., A lightweight web-based case tool for UML class diagrams, in Proceedings of the Fourth Australasian user interface conference on User interfaces 2003 - Volume 18. 2003, Australian Computer Society, Inc.: Adelaide, Australia. p. 95-98.
58. Maplesden, D., Hosking, J.G., and Grundy, J.C., A Visual Language for Design Pattern Modelling and Instantiation, in Design Patterns Formalization Techniques. 2007, ToufikTaibi (Ed), Idea Group Inc.: Hershey, USA.
59. Maurer, F., et al., Merging project planning and Web enabled dynamic workflow technologies. *Internet Computing*, IEEE, 2000. 4(3): p. 65-74.
60. McIntyre, D.W., Design and implementation with Vampire, in *Visual object-oriented programming*. 1995, Manning Publications Co. p. 129-159.
61. Mehra, A., Grundy, J.C., and Hosking, J.G., A generic approach to supporting diagram differencing and merging for collaborative design, in 2005 IEEE/ACM Automated Software Engineering. 2005: Long Beach CA.
62. Minas, M., Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 2002. 44(2): p. 157-180.
63. Minas, M., Visual Specification of Visual Editors with VisualDiaGen, in Applications of Graph Transformations with Industrial Relevance, J.L. Pfaltz, M. Nagl, and B. Böhlen, Editors. 2004, Springer Berlin / Heidelberg. p. 473-478.
64. Minas, M., Generating Meta-Model-Based Freehand Editors, in Electronic Communications of the EASST, Proc. of 3rd International Workshop on Graph Based Tools (GraBaTs 2006), Satellite event of the 3rd International Conference on Graph Transformation. 2006: Natal, Brazil.
65. Moody, D.L., The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE TSE* 2009.
66. Pei, Y.S., Hosking, J.G. and Grundy, J.C. Automatic Diagram Layout Support for the Marama Meta-toolset, In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing, Pittsburgh, USA, Sept 18-22 2011, IEEE Press.

67. Phillips, C., et al. The design of the client user interface for a meta object-oriented CASE tool. in Proceedings of Technology of Object-Oriented Languages, 1998. TOOLS 28. 1998.
68. Plimmer, B. and Apperley, M., INTERACTING with sketched interface designs: an evaluation study, in CHI '04 extended abstracts on Human factors in computing systems. 2004, ACM: Vienna, Austria. p. 1337-1340.
69. Qiu, L. and Riesbeck, C.K., An incremental model for developing educational critiquing systems: experiences with the Java Critiquer. *Journal of Interactive Learning Research*, 2008. 19: p. 119-145.
70. Rekers, J. and Schuerr, A., Defining and parsing visual languages with layered graph grammars. *Journal Visual Languages and Computing*, 1997. 8(1): p. 27-55.
71. Robbins, J.E. and Redmiles, D.F., Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Journal of Information and Software Technology*, 2000. 42(2): p. 79-89.
72. Schuckmann, C., et al., Designing object-oriented synchronous groupware with COAST, in Proceedings of the 1996 ACM conference on Computer supported cooperative work. 1996, ACM: Boston, Massachusetts, United States. p. 30-38.
73. Sutcliffe, A., *The domain theory: patterns for knowledge and software reuse 2002*: Mahwah, N.J.: L. Erlbaum Associates.
74. Vlissides, J.M. and Linton, M., *Unidraw: A framework for building domain specific graphical editors*, in UIST'89. 1989, ACM Press. p. 158-167.
75. Wohed, P., van der Aalst, W. M. P., Dumas, M., terHofstede, A. H. M. and Russell, N. On the Suitability of BPMN for Business Process Modelling, *Lecture Notes in Computer Science*, 2006, Volume 4102/2006, 161-176.
76. Zhang, J., Lin, Y., and Gray, J., Generic and domain-specific model refactoring using a model transformation engine. *Research and Practice in Software Engineering*, 2005. II: p. 199-218.
77. Zhang, K., Zhang, D.-Q., and Cao, J., Design, construction, and application of a generic visual language generation environment. *IEEE TSE*, April 2001. 27(4): p. 289-307.
78. Zhao, C., Kong, J., and Zhang, K., Program Behavior Discovery and Verification: A Graph Grammar Approach. *Software Engineering*, *IEEE Transactions on*, 2010. 36(3): p. 431-448.
79. Zhu, N., et al., Pounamu: a meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, 2007. 80 (8).

John C. Grundy is a member of the IEEE and the IEEE Computer Society. He holds the BSc(Hons), MSc and PhD degrees, all in Computer Science and all from the University of Auckland, New Zealand. Previously he was Lecturer and Senior Lecturer at the University of Waikato, New Zealand, and Professor of Software Engineering and Head of Electrical & Computer Engineering at the University of Auckland, New Zealand. He is currently Professor of Software Engineering and Head, Computer Science and Software Engineering, at the Swinburne University of Technology, Melbourne, Australia. He is an Associate Editor of IEEE Transactions on Software Engineering, Automated Software Engineering journal, and IEEE Software. His current interests include domain-specific visual languages, model-driven engineering, large scale systems engineering, and software engineering education.

John Hosking is Dean of Engineering and Computer Science at the Australian National University. John completed a PhD in Physics at the University of Auckland before joining the academic staff there. He progressed through to the rank of Professor and had 6 years as Head of Department before taking up his role at ANU. John has research interests in software tools, software engineering, and visual languages and environments. He is a Fellow of the Royal Society of New Zealand and MemIEEE.

Karen Li conducted a PhD followed by a Post-doctoral research in the University of Auckland from 2004-2011, specializing in visual software development notations and techniques. Karen is now working in industry, practicing visual programming (in particular the use of IBM technology stack) for client software development, through which she is gaining more insights in visual software techniques.

Norhayati Mohd Ali received the PhD degree in computer science from the University of Auckland, New Zealand in 2011. She is a senior lecturer at the Information System Department, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. Her research interests include critiquing tools in software engineering, visualization in software engineering, software engineering education, and human-computer interaction.

Jun Huh has worked as a research assistant for Prof. John Hosking and Prof. John Grundy since 2005, specialising in model driven visual development. He has been heavily involved in developing prototypes using and for the Marama meta-tools. Jun is currently working as a researcher and developer in Visual Wiki project, which involves visualisation of web-based information and knowledges.

Dr. Richard Li (Lei) is the Enterprise Information Services Manager at Beef + Lamb New Zealand. This role directly reporting the Chief Operating Officer is to design the enterprise level information management strategy and lead the development of technical solutions. Richard's expertise predominantly in the Business Process Modeling and Technology Integration and Transformation, but also covers Human Computer Interaction, Visual Languages and Information Management Strategy.

2.5 WikiBuilder: end-user specification and generation of Visual Wikis

Hirsch, C., Hosking, J.G. and Grundy, J.C. WikiBuilder: end-user specification and generation of Visual Wikis, In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, Belgium, 20-24 Sept 2010, ACM, pp 13–22.

DOI: [10.1145/1858996.1859002](https://doi.org/10.1145/1858996.1859002)

Abstract: With the need to make sense out of large and constantly growing information spaces, tools to support information management are becoming increasingly valuable. In prior work we proposed the "Visual Wiki" concept to describe and implement web-based information management applications. By focusing on the integration of two promising approaches, visualizations and collaboration tools, our Visual Wiki work explored synergies and demonstrated the value of the concept. Building on this, we introduce "WikiBuilder", a Visual Wiki meta-tool, which provides end-user supported modeling and automatic generation of Visual Wiki instances. We describe the design and implementation of the WikiBuilder including its architecture, a domain specific visual language for modeling Visual Wikis, and automatic generation of those. To demonstrate the utility of the tool, we have used it to construct a variety of different Visual Wikis. We describe the construction of Visual Wikis and discuss the strengths and weaknesses of our meta-tool approach.

My contribution: Developed initial idea for the approach, co-designed the approach, co-supervised PhD student working on project, co-authored significant amount of the paper, co-lead investigator for funding for the work from Foundation for Research Science and Technology and BuildIT.

VikiBuilder: end-user specification and generation of Visual Wikis

Christian Hirsch, John Hosking
Department of Computer Science
The University of Auckland
Auckland, New Zealand

chir008@aucklanduni.ac.nz,
j.hosking@auckland.ac.nz

John Grundy
Faculty of Information & Communication Technologies
Swinburne University of Technology
Melbourne, Australia

jgrundy@swin.edu.au

ABSTRACT

With the need to make sense out of large and constantly growing information spaces, tools to support information management are becoming increasingly valuable. In prior work we proposed the “Visual Wiki” concept to describe and implement web-based information management applications. By focusing on the integration of two promising approaches, visualizations and collaboration tools, our Visual Wiki work explored synergies and demonstrated the value of the concept. Building on this, we introduce “VikiBuilder”, a Visual Wiki meta-tool, which provides end-user supported modeling and automatic generation of Visual Wiki instances. We describe the design and implementation of the VikiBuilder including its architecture, a domain specific visual language for modeling Visual Wikis, and automatic generation of those. To demonstrate the utility of the tool, we have used it to construct a variety of different Visual Wikis. We describe the construction of Visual Wikis and discuss the strengths and weaknesses of our meta-tool approach.

Categories and Subject Descriptors

D.2 [Software Architectures]: Software Architectures; H.3 [Information Storage and Retrieval]: Information Search and Retrieval; H.5 [Information Interfaces and Presentation]: Hypertext/Hypermedia.

General Terms

Design, Documentation, Human Factors.

Keywords

Visual Wiki, knowledge management, visualization, code generation, domain specific visual language, modeling.

1. INTRODUCTION

Wikis have become increasingly popular for collaboratively creating, managing and sharing knowledge. This includes both for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

Copyright 2010 ACM 978-1-4503-0116-9/10/09...\$10.00.

large scale, general purpose knowledge repositories, such as Wikipedia, but also increasingly for corporate usage [15]. Wikis have significant advantages in their open collaborative approach to (often tacit) knowledge capture and manipulation (wiki “gardening”). However, the popularity of wikis does lead to a problem with scale: as the size of wikis increase, users increasingly find it difficult to locate and assimilate the knowledge they need [3]. As a result, while establishing a “wiki culture” within an organization is readily achievable, getting people to sustain their usage of a wiki can be problematic due to these search and navigation issues. To mitigate these issues, we have been exploring the concept of a Visual Wiki [13]. This combines visualizations, providing a high level overview, and wiki pages, providing more detailed information juxtaposed in a focus-plus-context oriented format. Having successfully manually developed and evaluated a variety of applications based on our Visual Wiki conceptual model, we wanted to make the development of Visual Wiki style applications easier, ideally by end-users themselves. Accordingly we have designed and implemented VikiBuilder, a meta-tool supporting the specification and realization of Visual Wiki applications. We begin by motivating our research, including an introduction to our Visual Wiki conceptual model. We then provide a high level description of our VikiBuilder approach and the meta-tool realizing it. Following a more detailed description of the application’s architecture and implementation, we prove its utility by describing its use to implement a variety of Visual Wikis. This leads into discussion of our experiences using and evaluating VikiBuilder and we outline some areas for future research.

2. MOTIVATION

A Visual Wiki [13] is a web-application integrating a textual and a visual representation of the same underlying body of knowledge. Both or either of the representations may be editable in a shared, traditional wiki style. The purpose of a Visual Wiki is to increase the effectiveness of wikis as knowledge management tools, via visual enhancements. As shown in Figure 1 our Visual Wiki concept consists of four components: the problem domain, the textual and visual representation, and a mapping in between.

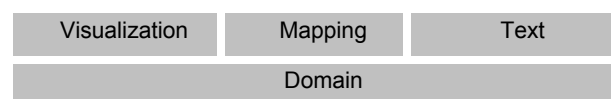


Figure 1. The four components of the Visual Wiki.

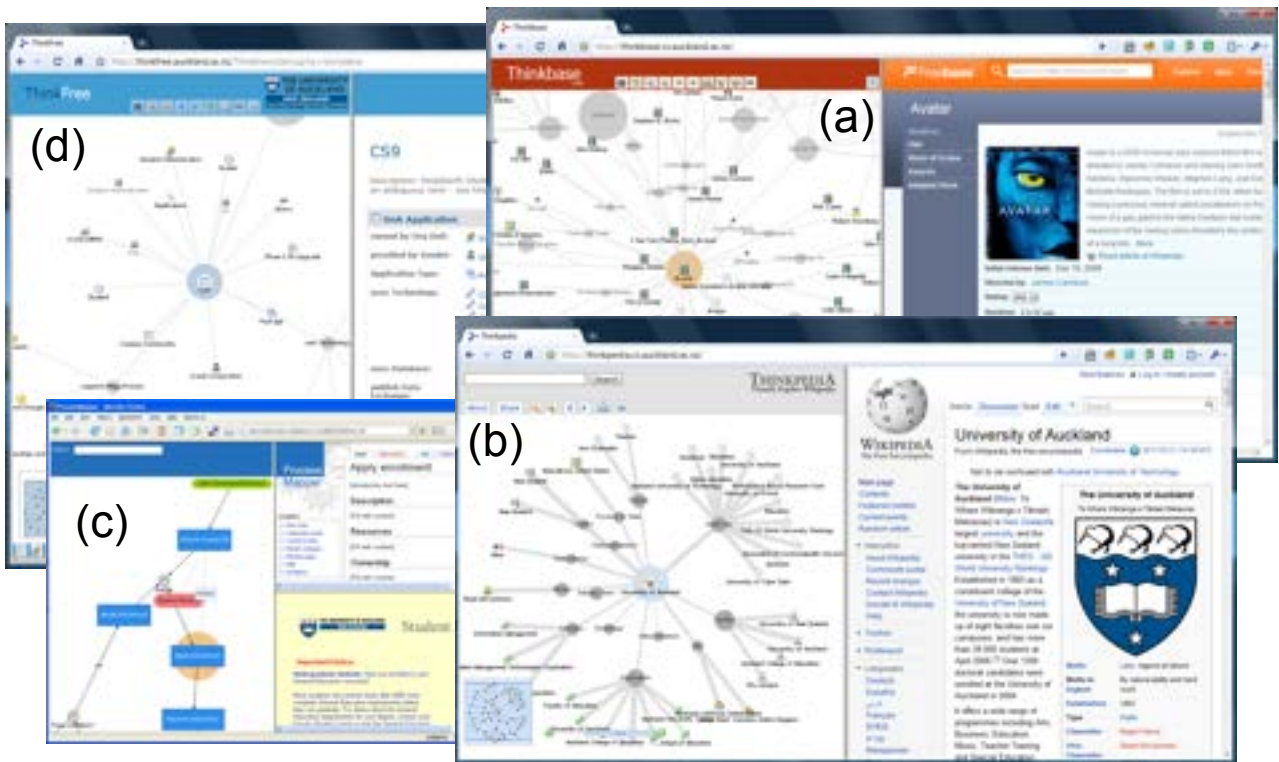


Figure 2. Visual wiki applications: (a) Thinkbase (b) Thinkpedia (c) ProcessMapper (d) ThinkFree.

The *domain* component describes the purpose and content of the Visual Wiki. For example, it could be meant for tasks such as search and exploration, or creation of information. The content specifies the problem domain information the application supports. The *text* and *visualization* components are similar and provide textual and visual interfaces to surface and interact with the domain content. The *mapping* component determines how the two representations are coordinated. This includes both the navigation behavior and consistency management policies. A more thorough discussion of our Visual Wiki concept can be found in [13].

Figure 2 shows several of our previously developed Visual Wiki applications. *Thinkbase*¹ (a) [11-13] is a visual navigation tool for *Freebase*², an open semantic wiki. The frame at the left shows a force-directed interactive graph of the relationships between the currently selected *Freebase* page (on the movie *Avatar*) and other semantic entities (nodes). Different icons represent different types of entities, aggregation nodes (in grey) collect entities related in a similar semantic way. Nodes can be expanded or collapsed. The frame on the right displays the *Freebase* wiki page for the currently selected topic. Users can navigate and explore the information space via the visualization. Additional search capabilities, accessible through context menus, are provided. We have also adapted *Thinkbase* for use in modeling and displaying software architecture documentation in

KaitoroBase [24]. *Thinkpedia*³ (b) [12] provides similar functionality for *Wikipedia*. As *Wikipedia* is less structured, we have used the *SemanticProxy*⁴ web service to “semantify” the wiki page content to produce the relationship graph. The width of edges specifies the strength of the semantic relationship as evaluated by the *SemanticProxy*. *ProcessMapper* (c) [13] visualizes business processes specified using *BPMN* and coordinates the visualization with wiki pages documenting process stages and organizational information and web applications realizing them. *ThinkFree* (d) is a Visual Wiki, deployed as an enterprise application, describing *Enterprise IT* assets at the *University of Auckland*. The asset descriptions are maintained in *Freebase*, but the visualization coordinates both, these descriptions as well as corporate wiki pages (in *Confluence*) and *SharePoint* documents relevant to an asset.

These applications, their evaluation and corporate deployment (in case of *ThinkFree*), have together demonstrated that the Visual Wiki concept provides considerable value for knowledge management [12, 13]. However, constructing each of the applications involves significant programming. While we have leveraged industrial strength components, such as the *Thinkmap*⁵ visualization toolkit, to reduce development, each application has still required substantial programming efforts.

¹ <http://thinkbase.cs.auckland.ac.nz>

² <http://www.freebase.com>

³ <http://thinkpedia.cs.auckland.ac.nz>

⁴ <http://semanticproxy.com>

⁵ <http://www.thinkmap.com/>

Many applications deal with visualizations of some kind of data or information and the display of those in (coordinated) multiple views. Much of this has focused on fields related to visualization (data, scientific, information) and much of it is based around dataflow through a pipeline of processing stages and tools that allow specification of processing elements (source selectors, filters, mappers, renderers) and pipelines connecting them. The “animation production environment” (apE) [21] and AVS [5] permit scientific visualization workflows to be specified and realized using direct manipulation visual programming interfaces. ConMan [9] and VTK [22] provide a similar approach for graphics applications. In information visualization, the “data state model” [4], which represents workflow as a series of data transformations, has been influential and has been adopted by popular information visualization toolkits such as Prefuse [10].

Coordinated Multiple Views (CMVs) have the premise that users understand their data better if they interact with the information and view it through different perspectives [20]. They add the need for a coordination model [2] to the dataflow to permit users to interact in a coordinated way across a range of different visualizations. Snap [18], has a more data-centric approach to coordination, but provides a direct-manipulation visual language for building visualizations. Similar approaches include: GeoVISTA [23], GeoAnalytics [14], and VisTrails [1].

A number of meta-tools have been developed over many years to enable rapid development of graphical design environments [7, 8, 16]. Typically these are desktop tools themselves and are used to generate desktop IDE-hosted visual design tools rather than web-based visualization tools. However, the concept of such visual specification of visual language tools has proved useful in this domain.

Other types of application using dataflow and visual languages for manipulation are mashup generators like Yahoo Pipes⁶ and (the now discontinued) Microsoft Popfly⁷. These applications allow various public feeds (e.g. RSS news feeds) and services (search engines, photo sharing sites, etc.) as data sources. Data manipulation uses operators such as filters or unions. The output consists e.g. of RSS feeds or mashups (for instance a map combined with geotagged images).

3. OUR APPROACH

To reduce the programming overhead of constructing Visual Wiki applications, we were motivated to develop a toolset through which we could straightforwardly specify and generate new Visual Wiki instances: the *VikiBuilder* meta-tool. We were attracted to the dataflow metaphor of much of the work described above as we felt this had a good closeness of mapping to the problem domain. We also felt the visual language based approaches, such as apE, AVS and Yahoo Pipes provided good productivity enhancement, and the base dataflow and coordination elements were appropriate for our domain.

Accordingly, we began by examining each of the Visual Wiki applications we had developed, abstracted from them a set of reusable generic processing elements, and defined “standard”

⁶ <http://pipes.yahoo.com>

⁷ <http://www.popfly.com>

APIs and parameters that could be used to integrate them together and customize them for specific purposes. Based on these generic elements, we derived a Domain Specific Visual Language (DSVL) for specifying combinations of elements with dataflow based workflow connections linking them together. An environment for this DSVL was then realized which supported the specification of Visual Wiki applications. From these specifications, code generators supported their realization reusing the generic elements and generated “glue code”. This VikiBuilder Visual Wiki meta-tool is itself realized as a Visual Wiki application, providing its specification DSVL and textual information to users as a Visual Wiki.

4. VIKIBUILDER

4.1 A DSVL for Visual Wikis

From analysis of a range of our Visual Wikis, the generic types of processing element we identified are:

- Data Source: a data base, web service, flat file, etc.
- Adapter: describes how a data source is accessed. For example a web service could be accessed via its API. In that case the data access module describes this API.
- Data Representation: describes how the data is represented after access (or transformation).
- Transformation Agent: describes criteria e.g. for filtering a data representation or for merging data sources or data representations.
- View: describes the views within the frames.
- Coordination Object: sits between a view and another element; describes how events impact related elements.

Each of these generic processing element types correspond to a visual language element in the VikiBuilder DSVL. Instances of these elements are connected via dataflow connectors to specify a Visual Wiki application. Figure 3 shows these various visual language elements, together with VikiBuilder DSVL specifications of two of our Visual Wiki applications.

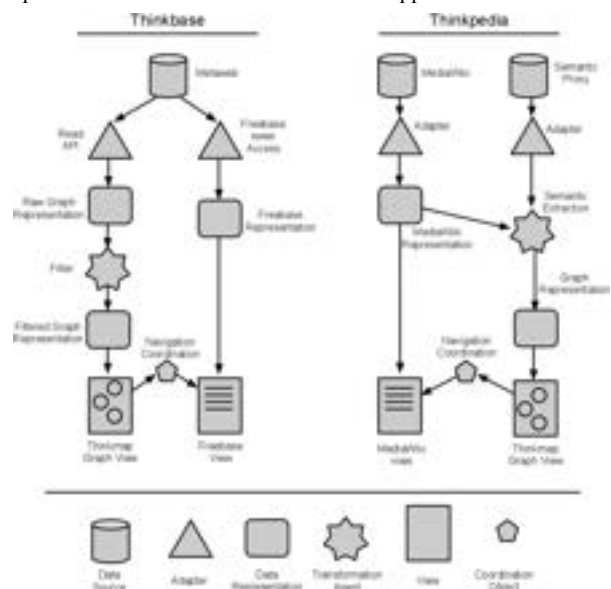


Figure 3. Thinkbase (left) and Thinkpedia (right) described using a dataflow metaphor.

At left is the *Thinkbase* specification. This has one data source, the Freebase (Metaweb) data base. A data access adapter element uses an API to access the source (left side of flow) resulting in a “raw” graph representation of the source. This is filtered using a transformation agent (e.g. filtering out specific node types) and the result displayed in a Thinkmap graph view (i.e. data in graph format is passed to Thinkmap to create a visual representation). On the right side of the flow is the standard web access to the Metaweb data through the Freebase view. A navigation coordination element specifies that a navigation event in the Thinkmap view influences the Freebase view (e.g. handing over the ID of the new center node object).

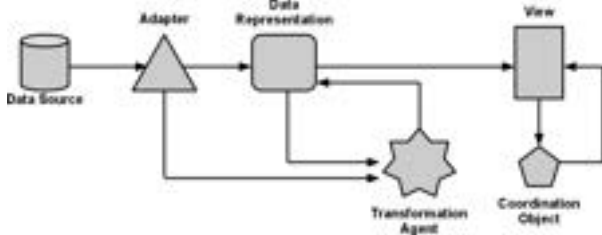


Figure 4. Overview of the VikiBuilder meta-model.

The *Thinkpedia* example in Figure 3 right uses two data sources, a MediaWiki and the SemanticProxy. A transformation agent specifies how the MediaWiki data is “semantified” using the SemanticProxy. The outcome is again a graph representation, which is displayed using a Thinkmap view. Additionally the

MediaWiki web representation is displayed in a second view. Generalizing from these and other examples, Figure 4 is an overview of the meta-model for the VikiBuilder DSLV, expressed using the same DSLV notation. For clarity, many details, e.g. element parameters and properties, are omitted.

4.2 VikiBuilder: a Visual Wiki meta-tool

Figure 5 shows our VikiBuilder meta-tool in use. The tool provides an environment for designing Visual Wikis using our DSLV, together with code generation and preview facilities that allow users to generate Visual Wikis from the specification and preview them as changes to the specification are made.

In Figure 5, VikiBuilder is being used to specify a Thinkpedia style tool using a visual specification similar to Figure 3 (right). A DSLV editor (1) allows modeling and visualization of the Visual Wiki design, with a tool bar (2) and project management facilities (3) at left. Details of the visual elements (specific element values, parameters, etc.) used in the DSLV model are specified in a Freebase view (4). The DSLV and Freebase views are coordinated, meaning the application is itself a Visual Wiki. As one can see we have had to make some compromises in the appearance of the DSLV (1) due to limitations in the Thinkmap visualization engine, e.g. use of icons instead of shapes (compare with Figure 3).

The modeling application uses Freebase as data storage and Thinkmap for the visual interface. The tool allows the creation, storage and editing of Visual Wiki architectures which are

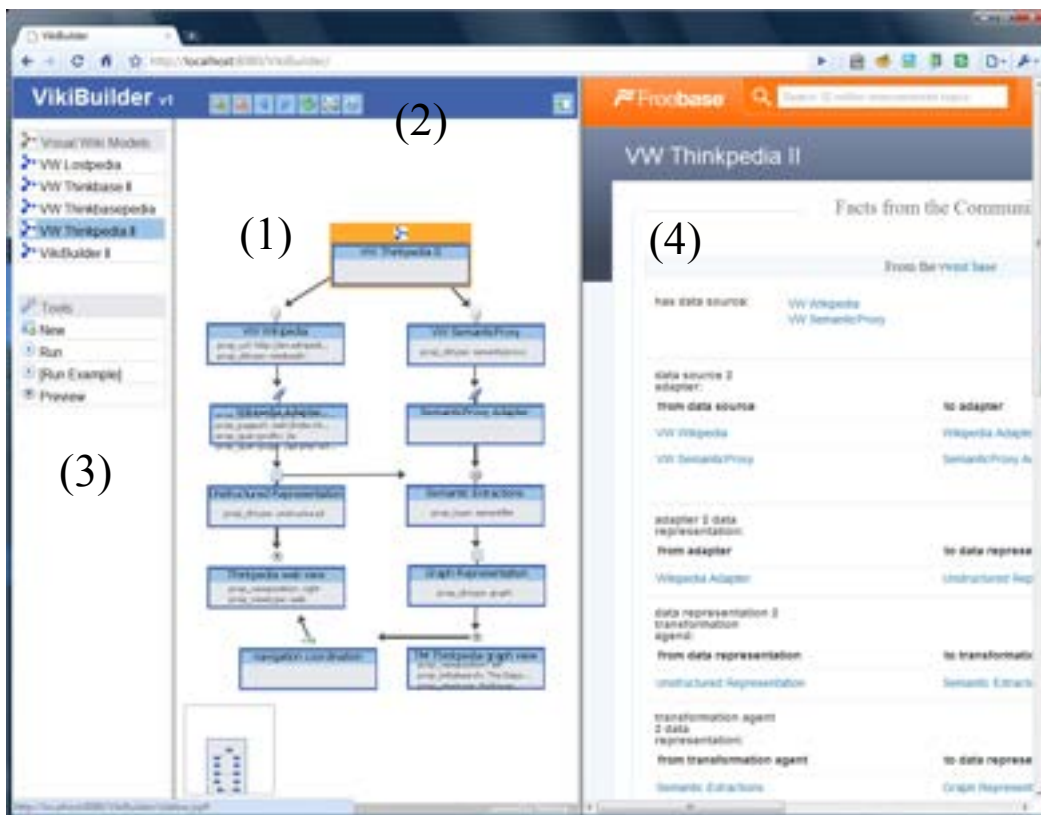


Figure 5. VikiBuilder meta-tool in use.

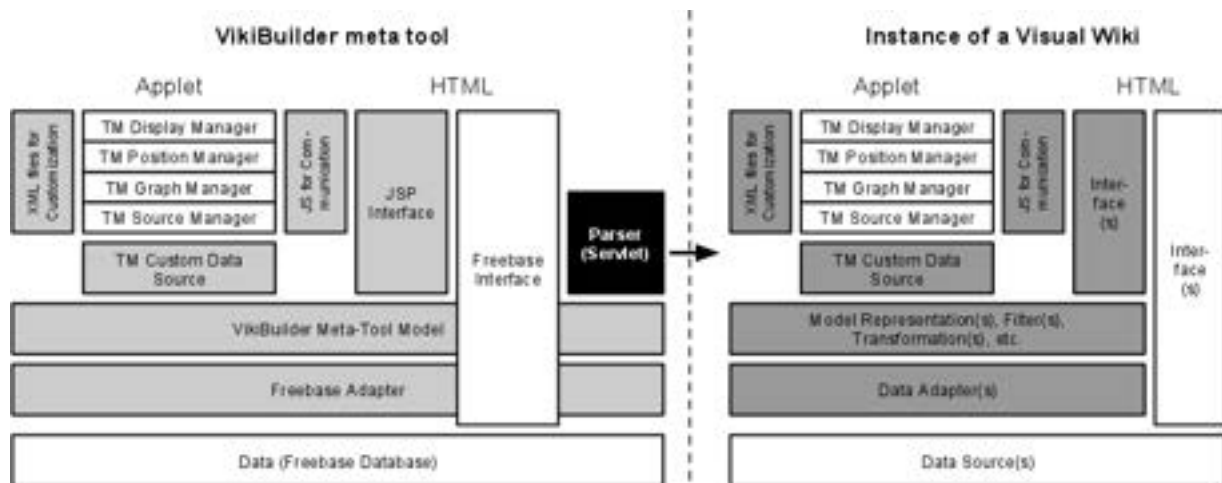


Figure 6. The production of a Visual Wiki instance (right) based on a model developed in the meta-tool (left).

stored in a Freebase domain. The schema of the Freebase domain represents the meta-model of the DSVL, elaborating the model shown in Figure 4. In designing the application our aim was to (1) make the application as easy to use as possible and (2) to use a metaphor for construction that would appeal to Visual Wiki designers. Constructing the application as a Visual Wiki itself supported both of these design aims.

Once a Visual Wiki application has been specified, the meta-tool can automatically generate the Visual Wiki instance. This takes the visual specification and compiles it to appropriate code and wiki templates to construct the application. The VikiBuilder tool also provides Preview support for generated Visual Wikis. In Figure 7 we can see the preview facilities in use. Here a Visual Wiki, similar to Thinkpedia, is being developed for the Lostpedia⁸ wiki, a wiki about the *Lost* TV show. The visual specification for the Visual Wiki is shown at left (1). The other design views (Freebase and toolbar) have been elided to provide screen space for the preview. The “VW Lostpedia” Visual Wiki preview is shown in the frames at the right. This Visual Wiki has a Thinkmap visualization at the top (2) and a Lostpedia wiki view at the bottom (3). The preview can be used to test out the design decisions made in the Visual Wiki design prior to deployment of the new Visual Wiki application.

4.3 Architecture and implementation

The functionality of the meta-tool can roughly be divided into two main areas or steps:

1. Functionality to allow a user to model a Visual Wiki. This includes an interface to create, change, and save models.
2. Functionality to transform this model into an instance of a Visual Wiki: process models (from 1) to automatically create code, property files, etc, and compile and present them.

Step 1 is similar to previous implementations (e.g. Thinkbase) and will be described first. This will also include a brief description of the architecture as well as the directory system, as those will be extended in step 2.

Figure 6 left shows the different layers of the pipeline architecture of the meta-tool (for step 1). From bottom to top: Freebase is the data source for the application. A customized “domain” in Freebase supports modeling Visual Wikis. The Freebase adapter makes use of the Freebase API and provides access to the data in a convenient way. The Visual Wiki meta-tool model is an internal representation of the model retrieved from the Freebase database. The model is turned into an interactive visual representation using Thinkmap (left side). This includes a custom data source layer, which translates the Model into the format required by Thinkmap. The remaining

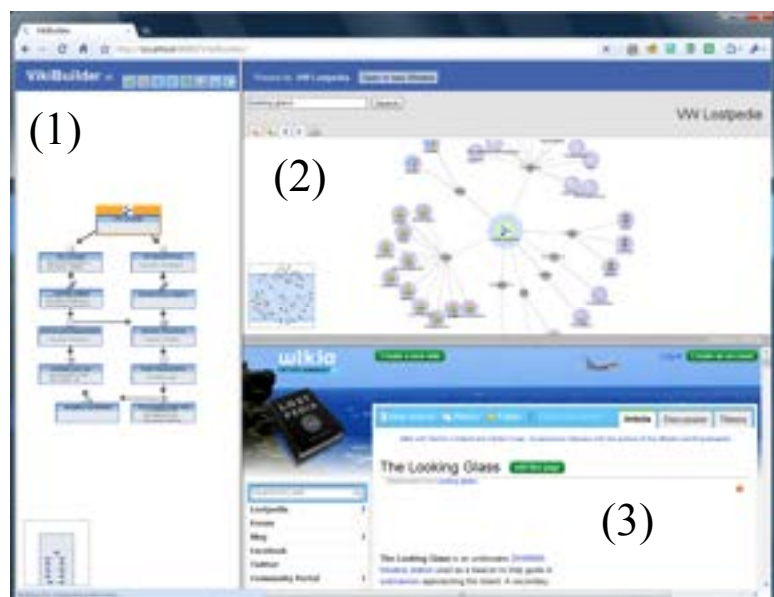


Figure 7. VikiBuilder preview for Lostpedia.

⁸ <http://lostpedia.wikia.com>

Thinkmap layers (Source, Graph, Position, and Display) are provided by the Thinkmap SDK. However, they are customized via XML files (e.g. their behavior and appearance). To the right, a JSP interface accesses the Visual Wiki meta-tool Model to create parts of the application's user interface. The default Freebase interface is currently used to edit the model.

Step 2 takes the model of a Visual Wiki (stored in Freebase) and turns it into an actual instance of a Visual Wiki. This is implemented using Servlets, which parse the model, create code and compile the new instance. A Parser Servlet (Figure 6, centre) is used to generate the new Visual Wiki instance (Figure 6, right) as follows:

1. The model is traversed and all the needed directories and files are created based on the properties of the model entities. These include: User interface specifications, including JSP and XML property files; back-end logic code such as data source access and data transformation; and an XML build specification file.
2. After all the necessary files have been created, an automatically invoked build tool uses the XML build file to construct the new Visual Wiki instance (compile and deploy the code, etc).

This process generates all of the darker components shown in Figure 6 (right) and links them to the other preexisting components, such as the data sources.

5. EXAMPLES

5.1 Thinkpedia II

In order to test the VikiBuilder meta-tool and prove the concept of our tool, we have modeled and created several Visual Wiki instances. As a first example, we have re-built our Thinkpedia prototype (see Motivation section) in the form of *Thinkpedia II*, i.e. a Visual Wiki, which has the same basic design features as one of our originally hand-crafted applications. As described in

Section 2, Thinkpedia is a visual exploration tool for Wikipedia. It uses the SemanticProxy service to extract semantically enriched concepts out of Wikipedia articles, visualizes those in an interactive graph, and therefore provides a visually appealing way to browse and explore the vast Wikipedia contents. To realize a re-implementation of Thinkpedia, we started by describing the model of the application in our VikiBuilder visual language. The final model (described theoretically in Figure 3) is shown in Figure 8.



Figure 8. The Thinkpedia II model created in VikiBuilder.

The key entities of the model are: Wikipedia and SemanticProxy as the data sources; adapters, which access them (through their respective APIs); a transformation agent, which combines the “raw” content from Wikipedia with the



Figure 9. Thinkpedia II (left) and the original hand-crafted Thinkpedia (right).

SemanticProxy; a resulting structured data representation; a Thinkmap view, which further processes and visualizes this data representation; a standard Wikipedia view; and a coordination object, which defines the navigation behavior. All of these entities have properties which describe the specifics of Thinkpedia II. For instance the Wikipedia data source and the adjacent adapter entities describe how the application makes use of Wikipedia. Properties include e.g. the URL of the wiki and the MediaWiki API (e.g. search API and its parameters). Properties of the view entities include, amongst others, definitions of the type of view (e.g. Thinkmap) and the positioning of the frames. The creation of this model can be done without any need for programming through the form-based editing interface of Freebase (see Figure 5).

After the model has been created, we can hit the “Run” button, and Thinkpedia II is automatically created as a new Visual Wiki instance. As described in Section 4.3, this is done by parsing the model which will automatically create code and property files based on the entities and their parameters in the model. Finally, the code is compiled. The outcome is a new stand-alone instance of a Visual Wiki, which is accessible through its own URL. The

application specification can be loaded into the VikiBuilder again and further refined.

Figure 9 shows the final Thinkpedia II Visual Wiki adjacent to our original hand-crafted Thinkpedia. The new Thinkpedia II can be used in just the same way as the original application. An interactive graph of a selected Wikipedia article is visualized next to that article. A user can explore and navigate the wiki space via the graph, by clicking on nodes, which will update the visualization as well as the wiki view. All the basic features, such as navigation, search, and browsing history are the same as in the original Thinkpedia. Some of the more advanced features of our customized Visual Wikis are not possible to specify and implement in the current version of the VikiBuilder. In the case of Thinkpedia II, these include the visualization of semantic relevance (reflected in edge thickness), advanced filtering mechanisms, and a sharing feature for the graph view.

5.2 Lostpedia Visual Wiki

After showing the feasibility of re-implementing one of our original prototypes, we explored modeling and automatically creating different variations of Thinkpedia II style Visual Wikis. One of these is *VW Lostpedia*, which can be seen in Figure 10.

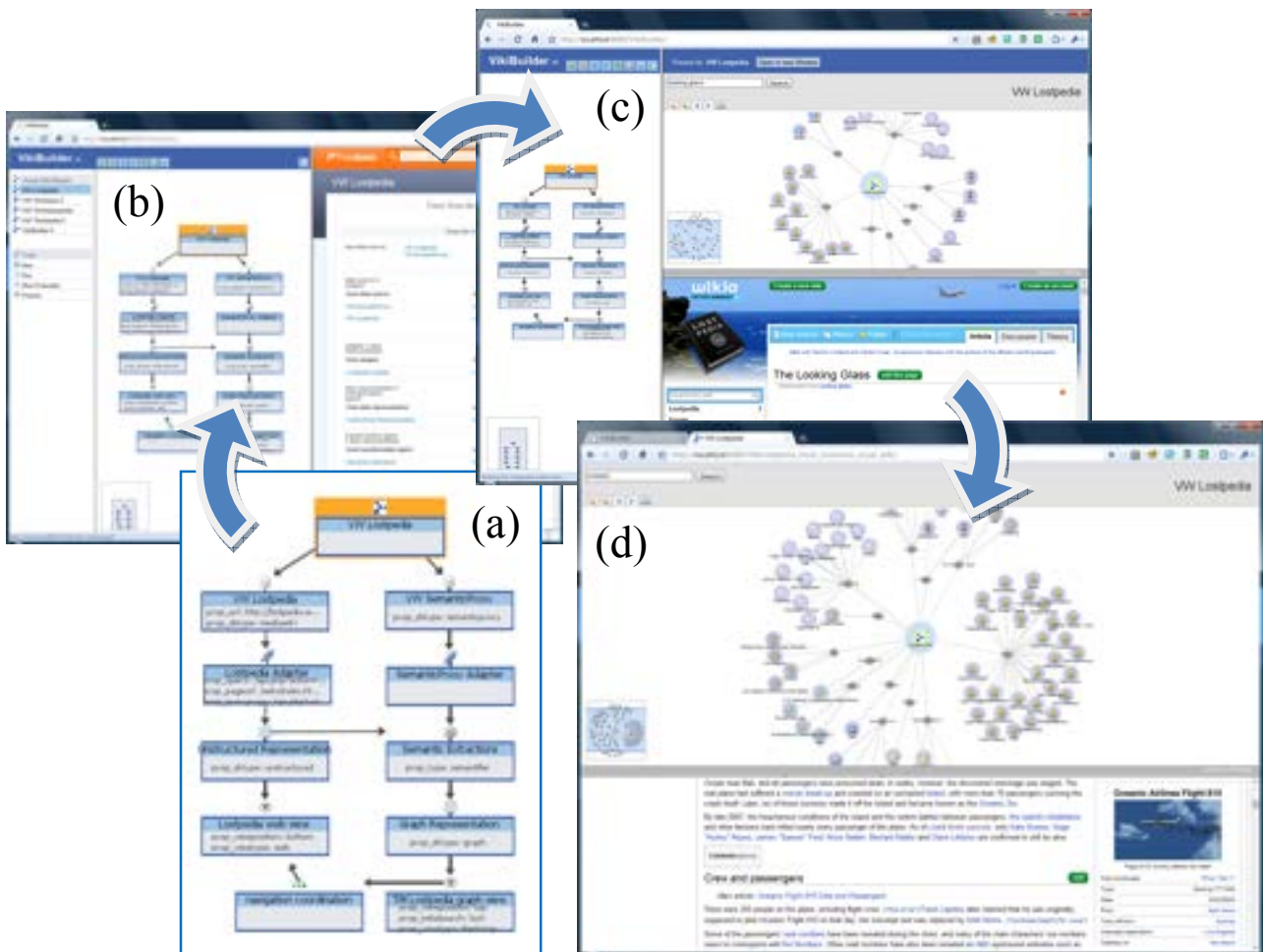


Figure 10. Lostpedia: The complete building process from (a) the model, to (b) the specification of the model, to (c) a preview of the Lostpedia Visual Wiki, to (d) the final stand-alone instance of the new Visual

VW Lostpedia is a visual exploration tool for Lostpedia, a wiki about the *Lost* TV show. We started modeling this Visual Wiki instance by taking the model from Thinkpedia II as a starting point: models can be copied and renamed to support simple reuse. Figure 10 (a) shows the final model for VW Lostpedia. It is similar to Thinkpedia II as it uses a MediaWiki-based wiki as a data source, extracts semantic meaning out of it utilizing the SemanticProxy, and visualizes information with Thinkmap. The main difference is that we are using a different wiki. As a user this can be realized by specifying different URLs and API parameters for the data source and adapter entities. Furthermore, we changed several parameters of the view entities so that they would be displayed in different frame locations. The complete production life-cycle of the VW Lostpedia can be seen in Figure 10: After specifying (a) and editing the model in the Freebase view (b), we can preview the application (c), and finally produce the stand-alone instance of a new Visual Wiki (d). By modeling and implementing VW Lostpedia, we have shown that we can also produce a new Visual Wiki rather than just replicating an existing one. Currently we are able to rapidly model and automatically create similar customized Visual Wiki variations with virtually any MediaWiki-based wiki in typically less than a day.

5.3 Thinkbase II

As a third example we re-modeled another one of our prior Visual Wiki implementations, Thinkbase (see Figure 2), which allows visual exploration of the semantic wiki Freebase by extracting topics and their semantic relationships using the Freebase API and visualizing them in an interactive graph representations. Our goal was to model and automatically create a Visual Wiki with the same basic functionality: *Thinkbase II*. We have created the necessary dataflow model for Thinkbase II inside the VikiBuilder, see Figure 11. The application has only one data source, the Freebase Metaweb database, which on the one hand is accessed through the standard Freebase user interface, and on the other side through its API. The raw graph which is retrieved from that API is first filtered (certain entities will be filtered out) and then handed over to the Thinkmap framework to create the visualization. Detailed parameter specifications in the case of Thinkbase II include the way the Freebase API is accessed, how the data format of the source is converted into an internal representation, and on what criteria entities are filtered out. Figure 12 shows the final Thinkbase II Visual Wiki instance. The application has the same basic features as the original hand-crafted version of Thinkbase. The Freebase contents can be explored visually by navigating along the graph. The filter element even provides a new and more rapid way of altering the appearance of the graph. For instance, Figure 12

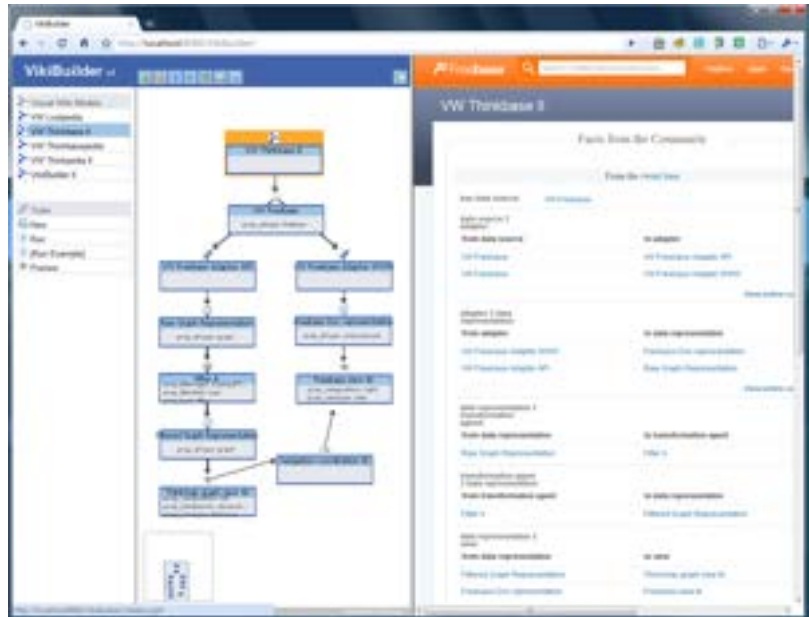


Figure 11. The Thinkbase II model in VikiBuilder.



Figure 12. Thinkbase II Visual Wiki.

shows the application where all entities which are not related to the type “Movie” are filtered out. However, there are also still some limitations compared to the original version. For instance, some specific types of nodes (e.g. URL nodes) are currently not displayed.

Our experience with these three exemplar Visual Wiki instances, which we are able to model and generate, has, we believe, successfully proven the concept of the VikiBuilder. We

were able to rapidly re-create previously hand-crafted Visual Wiki instances including all of their core functionalities, as well as new variations of them. Both our visual language to model Visual Wikis as well as the code generator are flexible and readily extendible, so that they can be easily refined in our future work to allow a broader range of Visual Wikis to be specified and generated.

5.4 VikiBuilder II

As a final example we attempted to model our own VikiBuilder (which is itself a Visual Wiki) inside the VikiBuilder meta-tool. The outcome of this *VikiBuilder II* model can be seen in Figure 13. As with our Thinkbase tool, the VikiBuilder II model specifies Freebase as its data source. On the one side, this data source is access via the API, split up into two flows, filtered using different criteria, and finally displayed in two different views, a menu view (for navigating and managing the different Visual Wiki models) and a Thinkmap graph view. On the other side, the Freebase content is shown using its default web interface. The three views are coordinated. In case of VikiBuilder II, the model is not yet able to automatically produce a finished Visual Wiki instance, as the VikiBuilder is much more complex compared to our other applications. This is especially the case for the code generation functionality, which we are not yet able to model in our VikiBuilder visual language. Extending our modeling language as well as the code generator in order to support these and similar advanced features will be part of our future work.



Figure 13. VikiBuilder II modeled inside VikiBuilder.

6. DISCUSSION

Our prior work developing hand-crafted Visual Wikis and their evaluations have successfully demonstrated the usefulness and appeal of the Visual Wiki concept [12, 13]. VikiBuilder provides the ability to develop Visual Wiki applications rapidly with minimal hand-crafting. This has been demonstrated by the success we have had in replicating implementations of our earlier Visual Wikis, together with additional new Visual Wiki applications, such as the Lostpedia Visual Wiki, using VikiBuilder. In each case, the time taken for implementation was of the order of days (or shorter), rather than the weeks or months of the prior applications. Rather than having to focus on technical programming issues, we were able to focus on more

creative issues, such as placement of frames and so on, with rapid feedback on changes to the design possible via the preview facilities.

We feel our decision to design VikiBuilder as a Visual Wiki itself was a valid one. This approach provides good “closeness of mapping” [6] to the pipeline metaphor that most information visualization designers use. In addition, as with most maturing frameworks, having identified a library of reusable computational artifacts, the bulk of the effort in constructing a Visual Wiki application is in configuring parameters for these components, instantiating them, and writing “glue code” to connect them together [19]. The choice of Freebase for underlying data representation and storage suited these needs well. The frame based representational model of this semantic wiki lends itself naturally to describing templates for parameterized artifacts, and instantiations of them in a readable, readily editable format as Freebase pages. The usual disadvantage of such an approach, the hidden dependencies [6] created by such a plethora of small artifact descriptions, is obviated via the context view of the visual language specification, which provides both an overview, and the primary mechanism for specifying and understanding the “glue code” connections between artifacts. This is an example of Moody’s Principle of Complexity Management, which advocates such a hierarchical decomposition of notational views [17]. The coordination mechanism between the visual and Freebase views supports Moody’s Principle of Cognitive Integration; providing explicit linkages between the different notational diagrams [17].

Weaknesses of our current VikiBuilder implementation include the following. Firstly, and most importantly, the range of Visual Wiki elements is currently limited. We would like to expand VikiBuilder to permit Visual Wiki implementations using other components, e.g. other visualization interfaces, such as Prefuse⁹, other semantifier services, various analysis services and additional data sources, such as adaptors for other wiki and database APIs. These are all straightforward programming tasks and will require minimal change to the existing tool architecture, and some minimal change to the code generation facilities.

Secondly, we have had to make some compromises in implementing the DSVL due to limitations in the Thinkmap visualization engine. For example, Moody argues that shape is one of the most important distinguishing characteristics for visual notational element design. We adopted this in the draft DSVL design, as seen in Figure 3, but were forced to use other visual channels, such as iconic annotations and color, in the Thinkmap realization. Other minor usability issues also result from such compromises. Expansion of the range of components noted above opens up the possibility of re-implementing VikiBuilder using itself, to effectively port VikiBuilder to other front-end technologies obviating some of these issues.

In future work, we plan to extend the range of components available to implement Visual Wikis. This includes incorporation of new instances of existing component types, as outlined above, but also extensions to the VikiBuilder DSVL to accommodate new types of element, such as a more refined ability to express and realize semantic analysis workflows, and a

⁹ <http://prefuse.org/>

broader range of annotations, such as the semantic strength link widths in the original Thinkpedia. These latter extensions are obviously more complex and will require extensions to the VikiBuilder meta-model, editing tools and code generation facilities. We also plan to ameliorate various minor usability issues we have identified in the existing implementation.

We are obviously keen to use VikiBuilder to construct new Visual Wiki applications. The success of our earlier Visual Wiki exemplars has led to strong commercial interest in our approach. To satisfy this interest, we need to be in a position to move from craft to production: VikiBuilder provides us the mechanism to do so. In addition we are keen to further explore the broader potential of the Visual Wiki concept and we see VikiBuilder as a platform for us to more rapidly realize this, in the same way that our work on meta-tools for Visual Language design and implementation [7, 8] have assisted us in that latter domain.

7. CONCLUSIONS

We have introduced VikiBuilder, a Visual Wiki application which permits rapid specification and realization of Visual Wiki tools. The latter combine visualizations, to provide a context overview and contextual navigation, with complex wikis, where individual wiki pages provide more focused detail on topics of interest. Our prior work has demonstrated the utility and appeal of such applications. VikiBuilder provides a meta-tool to more rapidly design and implement them. We have demonstrated the utility of VikiBuilder by recreating several of our prior Visual Wiki applications and implementing new ones. This showed a very significant productivity gain over hand-crafted solutions. By extending the range and type of components able to be instantiated we will extend the range of Visual Wiki applications we can support.

ACKNOWLEDGEMENTS

The authors acknowledge the assistance of the BuildIT Doctoral Scholarship fund and the FRST Software Process and Product Improvement project.

REFERENCES

- [1] Bavoil, L., et al., *VisTrails: Enabling Interactive Multiple-View Visualization*. In Proc. of IEEE Visualization, 2005.
- [2] Boukhelifa, N. and P.J. Rodgers, *A model and software system for coordinated and multiple views in exploratory visualization*. Information Visualization, 2: 258-269, 2003.
- [3] Buffa, M. and F. Gandon, *SweetWiki: Semantic Web Enabled Technologies in Wiki*. Proceedings of International Symposium on Wikis, 2006.
- [4] Chi, E.H., *A Taxonomy of Visualization Techniques using the Data State Reference Model*. Proc. of InfoVis, 2000.
- [5] Dyer, D.S., *A Dataflow Toolkit for Visualization*. IEEE Computer Graphics and Applications. 10: 60-69, 1990.
- [6] Green, T.R.G. and M. Petre, *Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework*. Journal of Visual Languages and Computing, 7:131-174, 1996.
- [7] Grundy, J.C., Hosking, J.G., Li, N. and Huh, J. *Marama: an Eclipse meta-toolset for generating multi-view environments*, Formal demonstration at the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008, ACM Press.
- [8] Grundy, J.C., Hosking, J.G., Zhu, N. and Liu, N., *Generating Domain-Specific Visual Language Editors from High-level Tool Specifications*. Proceedings of the 2006 IEEE/ACM International Conference on Automated Software Engineering, Tokyo, 24-28 Sept 2006, IEEE.
- [9] Haerberli, P.E., *ConMan: a visual programming language for interactive graphics*. ACM SigGraph Computer Graphics, 22:103-111, 1988.
- [10] Heer, J., S.K. Card, and J.A. Landay, *Prefuse: a toolkit for interactive information visualization*. ACM, 2005.
- [11] Hirsch, C., Grundy, J.C., and Hosking, J.G., *Thinkbase: A Visual Semantic Wiki*. Demo Session of the 7th International Semantic Web conference, 2008.
- [12] Hirsch, C., Hosking, J.G., and Grundy, J.C., *Interactive Visualization Tools for Exploring the Semantic Graph of Large Knowledge Spaces*. 1st Int'l Workshop on Visual Interfaces to the Social and the Semantic Web, 2009.
- [13] Hirsch, C., Hosking, J.G., Grundy, J.C., Chaffe, T., MacDonald, D., and Halytsky, Y., *The Visual Wiki: A New Metaphor for Knowledge Access and Management*. Proc. of the 42nd Hawaii International Conference on System Sciences, 2009, IEEE CS Press.
- [14] Johansson, S. and M. Jern., *GeoAnalytics visual inquiry and filtering tools in parallel coordinates plots*. Proc. of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems, 2007.
- [15] Majchrzak, A., C. Wagner, and D. Yates., *Corporate wiki users: results of a survey*. Proceedings of International Symposium on Wikis, 2006.
- [16] Mazanek, S., S. Maier, and M. Minas, *Auto-completion for Diagram Editors based on Graph Grammars*. Proc. of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE CS Press, 2008.
- [17] Moody, D.L., *The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering*. IEEE Transactions on Software Engineering, 2009.
- [18] North, C., et al., *Visualization schemas and a web-based architecture for custom multiple-view visualization of multiple-table databases*. Information Visualization, 2002.
- [19] Roberts, D. and R. Johnson, *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*. Proc. of PLoP, 1996.
- [20] Roberts, J.C., *State of the art: Coordinated & multiple views in exploratory visualization*. ETH, Switzerland, IEEE Press, 2007.
- [21] Schröder, F., *apE—the original dataflow visualization environment*. ACM SigGraph Computer Graphics, 1995.
- [22] Schroeder, W.J., K.M. Martin, and W.E. Lorensen, *The design and implementation of an object-oriented toolkit for 3D graphics and visualization*. IEEE Visualizations 1996.
- [23] Takatsuka, M. and M. Gahegan, *GeoVISTA Studio: A codeless visual programming environment for geoscientific data analysis and visualization*. Computers and Geosciences, 28:1131-1144, 2002.
- [24] Ting, M.S., Hirsch, C., and Hosking, J.G., *KaitoroBase: Visual Exploration of Software Architecture Documents*, Formal demonstration at ASE, 2009.

3

DSVLs and MDE for Software Requirements and Architectures

3.1 Aspect-oriented Requirements Engineering for Component-based Software Systems

Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, 1999 *IEEE Symposium on Requirements Engineering*, Limerick, Ireland, 7-11 June, 1999, IEEE, pp 84 - 91.

DOI: [10.1109/ISRE.1999.777988](https://doi.org/10.1109/ISRE.1999.777988)

Abstract: Developing requirements for software components, and ensuring these requirements are met by component designs, is very challenging, as very often application domain and stakeholders are not fully known during component development. The author introduces a new methodology, aspect-oriented component engineering, that addresses some difficult issues of component requirements engineering by analysing and characterising components based on different aspects of the overall application a component addresses. He gives an overview of the aspect-oriented component requirements engineering process, focus on component requirements analysis specification and reasoning, and briefly discuss tool support.

My contribution: Sole author ; developed all ideas, software, wrote whole paper.

Aspect-oriented Requirements Engineering for Component-based Software Systems

John Grundy

Department of Computer Science, University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

Abstract

Developing requirements for software components, and ensuring these requirements are met by component designs, is very challenging, as very often application domain and stakeholders are not fully known during component development. We introduce a new methodology, aspect-oriented component engineering, that addresses some difficult issues of component requirements engineering by analysing and characterising components based on different aspects of the overall application a component addresses. We give an overview of the aspect-oriented component requirements engineering process, focus on component requirements analysis, specification and reasoning, and briefly discuss tool support.

1. Introduction

As software systems become ever more complex, developers use new technologies to help manage development. Component-based systems are one example offering potential for better existing or third party component reuse, compositional systems development, and dynamic and end user reconfiguration of applications [1, 16]. Component-based systems build applications from discrete, inter-related software components, often dynamically plugged into running applications and reconfigured by end users or other components [10, 16].

While some processes, notations and tools used for traditional Requirements Engineering [12] are useful for component development, we have found deficiencies during development of component-based design tools [2, 3, 4]. Stakeholders are often not clearly identifiable when analysing component requirements, and include end users, developers, and other components. Components typically provide and require services to and from end users and other components, which can usually be classified by

different systemic aspects of an application. Traditional requirements capture techniques don't usually achieve this for individual components. The "requires" and "provides" relationships are captured by notations like Object-Oriented Analysis (OOA), but with insufficient detail. A key aim of software components is to allow components to be interchangeable, but traditional analysis techniques don't adequately identify and describe generic interfaces for extensible user interfaces, persistency, distribution and collaborative work. Lastly, a suitable codification of requirements is needed by end users and other components at run-time. We have developed *aspect-oriented component engineering* using the notion of "aspects" of a system (e.g. user interface, persistency and distribution, user configuration, collaborative work), for which components provide or require services. Aspects help identify, categorise and reason about component requirements.

Section 2 motivates our work using an example application and deficiencies with traditional RE methods and existing component-based methods and tools. Sections 3 to 5 overview the concepts and process of aspect-oriented component engineering, illustrating requirements specification for our example application and discuss reasoning with aspect-oriented requirements. Section 6 describes how aspect-oriented requirements are used during component design, implementation and deployment, and discusses tool support for aspect-oriented component engineering. We conclude with a summary of contributions and overview of future research directions.

2. Motivation

Our need for improved component requirements engineering grew from experiences developing multiple view, multiple user design environments. An example of such a system, Serendipity-II, is shown in Figure 1 [2].

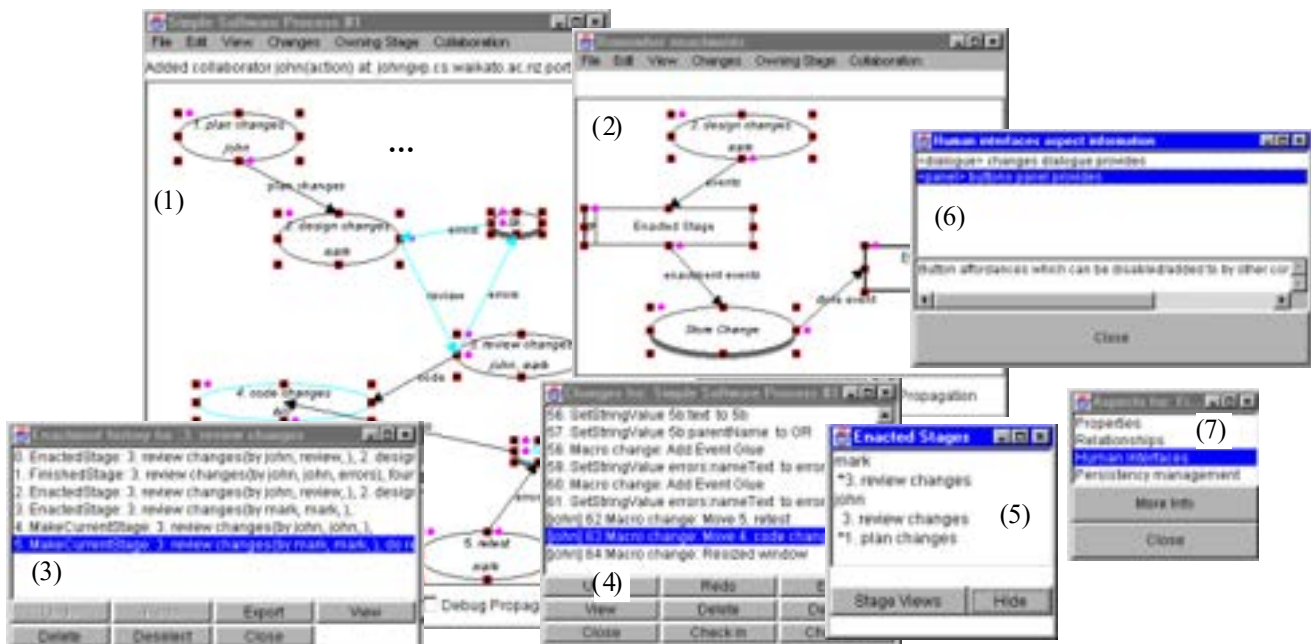


Figure 1. An example component-based application: the Serendipity-II process-centred environment.

Serendipity-II's main functional requirements include visual, collaborative process modelling and software agent specification views (1, 2), process enactment and view modification histories (3, 4), to-do lists (5), and agent component information (6, 7). Key non-functional requirements include supporting novice and experienced users, platform independence and mobile computer support, robustness, and security [4].

Serendipity-II was developed using component-based techniques, with many of the components making up the environment reused elsewhere. Examples include enactment and editing event history management and interfaces, collaborative view editing, persistency management, event broadcasting between environments, and version control and configuration management.

We found that traditional approaches to Requirements Engineering [12] are not ideal for developing component requirements. They tend to assume stakeholders and requirements are known, most parts of a system are used in one application (or are not dynamically configurable), and end users don't significantly reconfigure applications. Unfortunately existing component-based development methods usually focus on component design and implementation [1, 15, 16], and usually only provided services are documented. We found this leads to less-reusable components, particularly with regard to component user interfaces and support for distribution and collaborative work. Some have suggested provides-relationships between components be reasoned about [13, 17], though have focused on low-level interface specification. Determining customer requirements for product development [8] shares similar issues to component engineering, with stakeholders and

usage not well-known. Techniques of ensuring diverse specifications are consistent, use of multiple perspectives and careful refinement of requirements to designs could thus be useful in component RE. Aspect-oriented programming (AOP) [7] uses systemic "aspects" of objects (particularly data distribution and concurrency), augmenting traditional object classes and "weaved" into code. We view weaving from an inter-component view, rather than intra-method, with some components unchangeable COTS parts.

Existing component development tools, such as Visual Age™ [3], focus on design and implementation, as do many CASE tools, such as Rational Rose™ [14] and Software thru Pictures™ [6]. We have found such tools unsatisfactory for analysing and documenting component requirements. Similarly, the component characterisation used by component architectures, like JavaBeans [10] and COM [15], are too low level for describing requirements. Enterprise JavaBeans use a high-level service framework, though currently focus only on service provision. Tools supporting component deployment [7, 18], lack high-level information about components, making run-time configuration difficult. Similarly, most component repositories [11] utilise indexing mechanisms that don't adequately characterise components for retrieval and reuse.

3. Component Engineering with Aspects

We have been developing an aspect-oriented component engineering methodology. Aspect-oriented Component Requirements Engineering (AOCRE) within this focuses on identifying and specifying the functional

and non-functional requirements relating to key “aspects” of a system each component provides or requires. For example, a developer may identify user interface, collaborative work and persistency-related functional and non-functional aspects of a component, and document provision and required services of the component for each such aspect. Aspects may be decomposed into aspect “details”, for example the data transfer, event broadcasting and version management provides/requires aspect details for collaborative work support. We have developed some useful categorisations of component aspects for design environments, in Table 1. While these categories have been useful for systems we have developed, other categorisations may be better for other domains. Domain-specific aspects can also be identified for specialised components e.g. process modelling for Serendipity-II, which we have found useful for documenting and reasoning about domain-specific component characteristics.

Aspect	Aspect Details	Description
User interface	Views Affordances Feedback Extensible parts	Aspects supporting or requiring user interface, including extensible & composable interfaces for several comps
Collaboration	Sync. editing Versioning Locking protocol Awareness	Aspects supporting or required for collaborative work by users
Persistency	Save/load data Find data Locking Versioning	Aspects supported or required for data persistency management
Distribution	Obj. Identification Oper. Invocation Transaction Man. Robustness	Aspects supported or required for distributed object management
Configuration	PEMs & Aspects Property sheet Wizard	Aspects supported or required for end user or dynamic configuration of component

Table 1. Some useful component aspects.

Some components may have many aspects and others a few. Unlike traditional object-oriented analysis object services, aspects may share component services, required aspects are as important to characterise as provided aspects, and often more than one other component may provide or require a component's aspects. These “overlapping” aspects are a natural consequence of high-level categorisation of the systemic properties of components, and help requirements engineers gain understanding of related component characteristics.

We thus view aspect characterisation as a way to take multiple, systemic perspectives onto components, and thus better understand and reason about component data, functionality, constraints and inter-relationships. Note that some systemic aspects identified for a component may be redundant in some usage scenarios, and different aspect categorisations may be used depending on both the aspects of reused components and those identified for the system as a whole being developed.

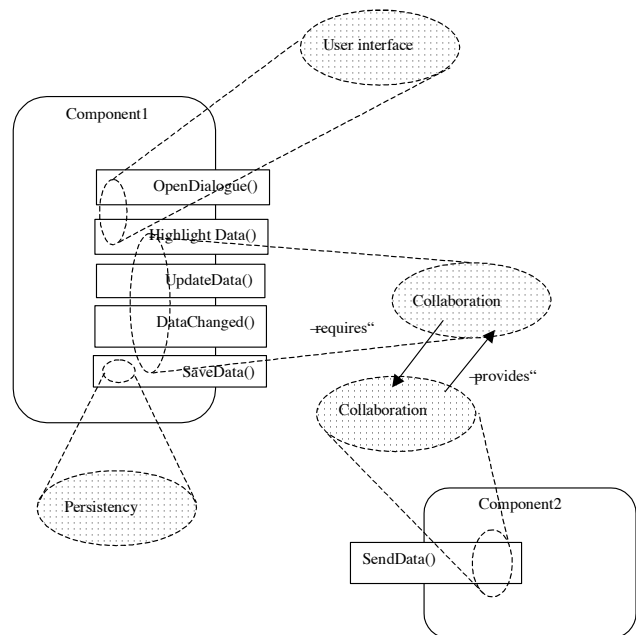


Figure 2. Basic notion of component aspects.

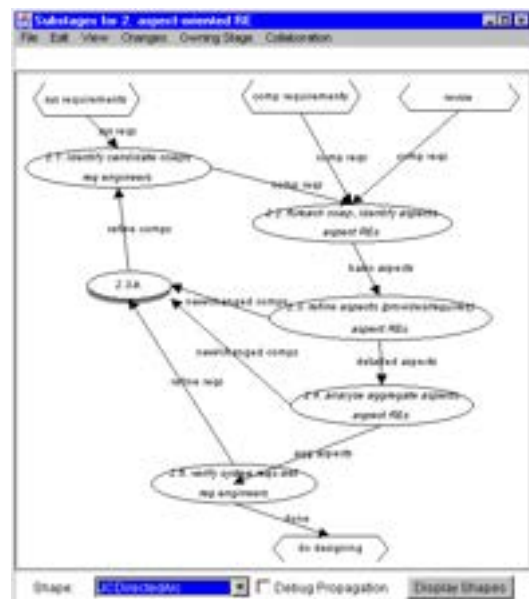


Figure 3. Basic AOCRE process.

Figure 3 shows the basic AOCRE process, which begins after analysing general application requirements or individual or groups of components requirements. This allows iterative top-down and bottom-up requirements refinement. Engineers characterise a component's aspects, aspect details, provided and required details, functional and non-functional properties, and reason about inter-related components' aggregate aspects. Components and aspects identified are refined into detailed component designs. For Serendipity-II we analysed requirements for some reusable components, then designed and implemented these. Serendipity-II requirements were

developed and refined into components and aspects. Aspects were reasoned with to determine component composition, configuration and reuse scenarios. Component design and implementation was carried out using these requirements, with feedback evolving reusable and Serendipity-specific requirements.

4. Describing Requirements Aspects

Candidate components are found from OOA diagrams, by reverse engineering software components, or bottom-up consideration of individual, reusable components. We have found "perfect" identification of components is not essential during AOCRE, with requirements-level "components" acting as groupings of related services and aspects. These can be split, merged or otherwise refined at design-time, in a similar way to OOA objects being refined into classes. For each component, we identify (using possible stakeholder requirements and object services) aspects for which the component provides services or requires services from other components.

For example, consider the event history component used in Serendipity-II, reused to provide view editing histories, processes stage enactment histories and collaborative editing histories of exchanged events. This component is identified from Serendipity-II requirements, which call for various event histories, or can be considered in a bottom-up fashion as a commonly required design environment component. Event history functional requirements include event management (add, remove, annotate), history display and manipulation,

multiple user sharing, and data persistency. Components may need to reconfigure event history user interfaces to enable/disable affordances or add their own (see Figure 1).

Figure 4 illustrates aspect-oriented requirements we have identified for the event history component, and some related components used with in Serendipity-II. Components are in solid rectangles, aspect characterisations in dotted rectangles. Aspect details are categorised as being "provided" by a component (denoted by a "+" prefix, e.g. dialogue, basic event management, data serialisation for the event history, or "required" ("0"- prefix), e.g. extensible affordance, event broadcasting and data storage. The aspects provided by the event history are shown in Figure 4, and the usage of provided aspect details and provision of required aspect details indicated between aspects and other Serendipity-II components.

When considering aspects for the event history we identified it must provide a user interface, provide collaborative work support, must be made persistent, and allow configuration of history behaviour. We made user interface affordances "extensible" by other components, avoiding a common problem of inconsistent user interfaces built from mis-matched parts. This need for extension was identified during Serendipity-II requirements specification, where a reused versioning component needs to extend event history affordances. We identified that collaborative work support infrastructure should be provided by other components, as these facilities are reused often by applications.

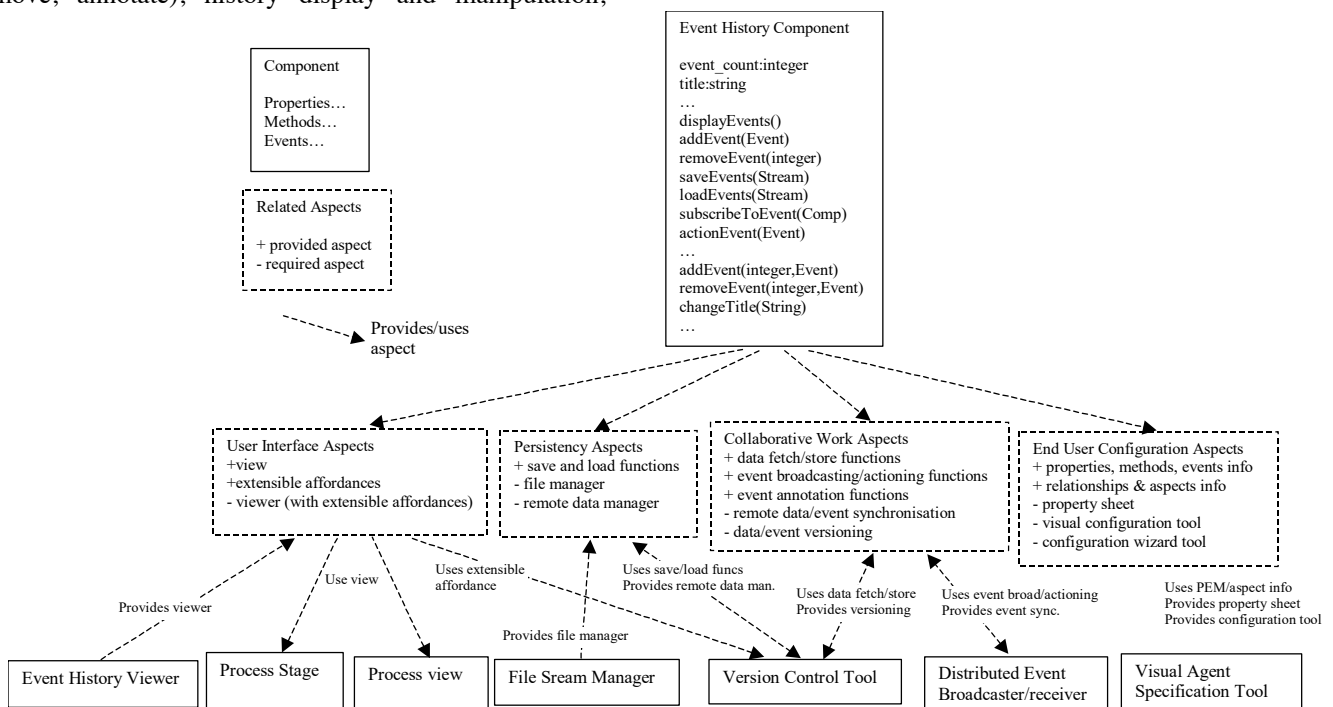


Figure 4. Example Serendipity-II components and some of their aspects.

II. Collaborative Work Aspects : COLLABORATION

II. 1) +data fetch/store functions : DATA_MANIPULATION
 -- Provides services for getting some/all of event history data and for updating some/all of event history data. Used by components providing collaborative work infrastructure to keep distributed data synchronised or partially synchronised.
 QUERY=true; UPDATE=true

II 2) +event broadcasting/actions functions : EVENT_MANAGEMENT
 -- Provides services allowing other components to detect event history update events and to action (replay) events received by other components. Used by components providing collaborative work infrastructure to keep distributed event history synchronised or support deltas of event history version changes.
 DETECT=true; ACTION=true

II 3) + event annotation functions : AWARENESS
 -- Provides services for annotating, selecting, highlighting events. Used by components providing collaborative work infrastructure to support basic group awareness facilities for updated event history events. Other components should use these to annotate events with remote user name, colour them with a colour associated with a particular user, etc.
 HIGHLIGHT=colour; ANNOTATE=text

II 4) - remote data/event synchronisation : LOCKING
 -- Requires component(s) that supports remote data/event synchronisation. Could support fully synchronised data or semi-synchronous update. This should be robust if network connections fail, and should work over low or high bandwidth networks.
 SYNCHRONOUS=true OR false; SEMI_SYNCHRONOUS=true OR false; NETWORK_SPEED=any; STORE=true

II 5) - data/event versioning : VERSIONING
 -- Requires component(s) providing data versioning. Should support both event history data and event history update event recording/versioning. This should be a simple-to-use facility for end users. Should extend the viewer affordances to provide at least check-in/check-out capabilities via +extensible affordance aspect.
 DATA=true; EVENT=true; INTERFACE=extensible affordances; CHECKIN=true; CHECKOUT=true

Figure 5. Detailed aspect-oriented component requirements specifications.

The event history provides basic collaborative work facilities, such as event editing, annotation, actioning received events and providing event listening and export facilities. It requires event and data broadcasting between environments and versioning facilities. Aspect details are kept quite general at the requirements level, and the eventual implementation of these facilities is generally unimportant. During AOCRE generalised aspect details are specified to characterise event history collaborative work-related services. Note event serialisation and deserialisation services are used by collaborative work and persistency aspects, illustrating aspects may overlap.

Detailed textual specifications of aspects provide additional documentation of functional and non-functional requirements. We are developing a set of properties for each aspect detail kind used to more formally describe aspects and aspect usage. Figure 5 shows an example of some codified aspect information for the event history.

5. Reasoning with Aspects

After identifying a component's provided and required aspects, related components and aspects can be reasoned about. Inter-component relationships inferred by provided and required aspects allow Engineers to reason about the validity relationships and aspects specified. For example, an event history linked to a component providing only event broadcasting collaborative work aspect doesn't have versioning. The component could be used but would not provide end users or the target application versioned event histories. If versioning is mandatory, the specification is invalid. If a history requires high bandwidth, encrypted data transfer, and is linked to a component providing only modem connection and no encryption, this is invalid.

Aggregate aspects can be identified and specified for groups of interrelated components, allowing Engineers to reason about aspect-oriented requirements for a set of related components, or even global requirements for a whole application. Figure 6 shows an example of a group of interrelated components providing an event history with asynchronous collaboration (via version control), persistency using files, and no synchronous collaborative support or extensible user interface. The aspects of this aggregate are a constrained subset of those of the event history and related components. Global application requirements can be specified using aspects, and then be migrated down to groups of related components or individual components.

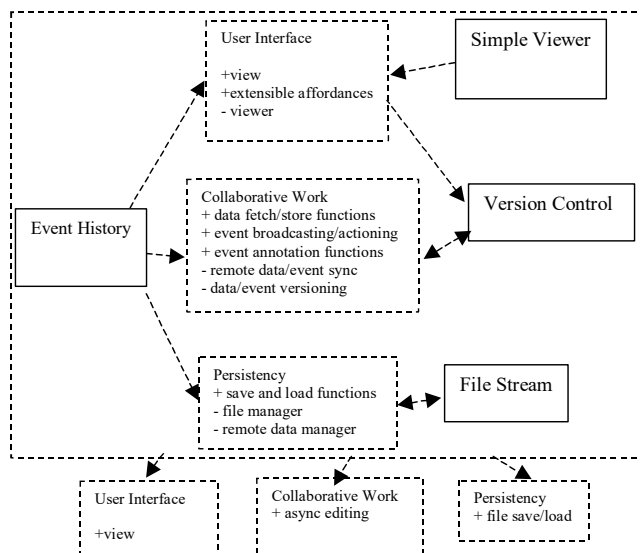


Figure 6. Example of aggregated aspects.

Aspects aid in handling evolution of requirements by assisting in categorising requirements changes and

localising effects of these changes to relevant aspect categories and aspect details. Changing overall system requirements impacts on aspect-oriented requirements specifications by: changing properties associated with aspect details, adding details or removing details; introducing new aspects and aspect details, or changing inter-component relationships and aspect provides/requires associations; and introducing new components or refining candidate components (merged, split), with modification of associated aspects. Aspects assist in reasoning about modified requirements by aiding requirements engineers in reformulating components, component aspects and provided/required aspects.

6. Design, Implementation and Run-time

Aspect-oriented component requirements assist when designing and implementing components. They provide a focused set of functional and non-functional constraints a design can be refined from, and provide a specification that an implementation can be tested against. Requirements-level components can be refined directly to matching design-level software components, or can be split, merged or otherwise revised, as can requirements-level component aspects. They also allow for design decisions to be influenced by weakening or strengthening aspect-level constraints.

Detailed design decisions about the user interface design and behaviour, component persistency and distribution strategies, technologies and available services, collaboration and awareness support facilities, and component configuration tools are usual refinements. Figure 7 shows an example of the refinement of event

history component requirements-level aspects to more detailed design-level software component aspects. Some aspects become more specific as e.g. user interface design decisions are made.

Aspects can provide a standardised mechanism for related components to describe and access each other's functionality, or be used to guide inter-component interface definition. A component may thus indirectly invoke other component functionality via operations provided by aspect implementations, or may invoke component operations directly. The former results in more generic, reusable inter-component relationships, while the later is sometimes easier to implement. Aspects can be implemented via interfaces, language reflection or design patterns. We have used all three approaches when implementing components with JViews, our component-based software architecture [8].

Aspect information can be encoded in component implementations for use at run-time by components or end users. Components may query other components for the aspects they provide or require, ask them to perform consistency checks for a configuration, or use their aspect information to reconfigure themselves. For example, a version control tool component queries the event history component for its user interface aspects, locates its preferred extensible affordance aspect (if there is more than one) and requests Check-in and Check-out affordances be added, and is notified when these are accessed. End users can peruse encoded aspect information to determine what functional and non-functional requirements a component has, as shown in Figure 1 (6, 7).

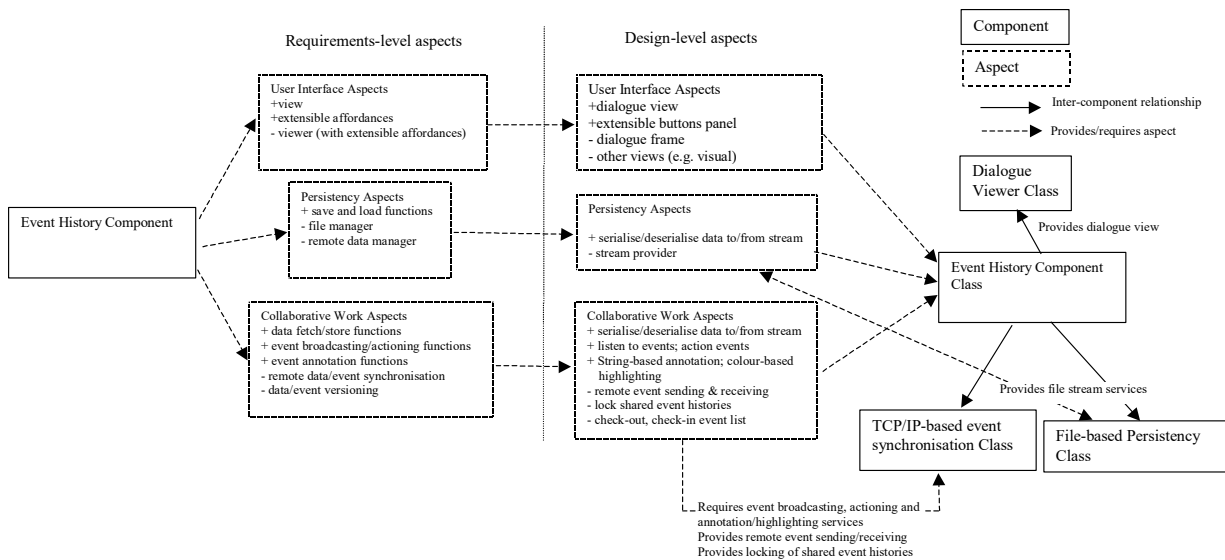


Figure 7. Refining requirements-level component aspects into design-level aspects.

7. Tool Support

We have developed some basic tool support for specifying aspects of components in a component-based software development environment, JComposer [3]. Aspects of a software component are grouped and associated with the component, and inter-component aspect usage documented. Some basic validation checking ensures related component aspect requirements are correctly met. Detailed aspect specifications are specified using a hierarchical notation in MS Word™ and aspect detail properties in JComposer. Basic inconsistency management techniques help manage evolving aspect-oriented requirements and include highlighting of changed aspect information in views, change histories for all views and each individual component and each of its aspects, and consistency checks that test if provide/require links between components match.

Requirements-level aspects can be refined into design-level aspects that are associated with classes used to implement a component. JComposer supports generation of information describing a component's aspects that developers and end-users can access at runtime. Basic reverse-engineering capabilities allow components with aspect information to be reverse engineered in JComposer, preserving their aspects.

Aspect-oriented Requirements Engineering can be used to analyse and refine the requirements of new or COTS components. We have used JComposer to characterise the aspects of various software engineering and office automation tools, including MS Excel™, MS Word™, Eudora™, JComposer itself, and Netscape™,

and these have been integrated with Serendipity-II [8]. When characterising the aspects of such third party components, it is only necessary to characterise those services or requirements of these systems that are to be used with other components.

To date we have reengineered many Serendipity-II and JViews components and developed several new components using our aspect-oriented approach. Requirements for these components have been documented using aspects and code part-developed using JComposer's support for design-level refinement of aspects. Previously we had used conventional requirements engineering and design approaches when developing environments like Serendipity-II. Our preliminary experience with AOCRE has been very positive, with improved documentation and understanding of component requirements resulting, along with an improved ability to reason about related component requirements using our aspect-oriented perspective. Generally we have found components that have been developed using AOCRE exhibit improved reusability and extensibility, and systems built with these components exhibit improved allocation of responsibility for data and behaviour among both reused and application-specific components.

We are exploring additional visual language support for aspects and aggregated aspects, including better indication of provides/requires aspects spanning several components. Extending our property/value aspect descriptions will help better-describe aspects and provide more formal, rigorous checking. We are developing a repository using aspects to index components for reuse.

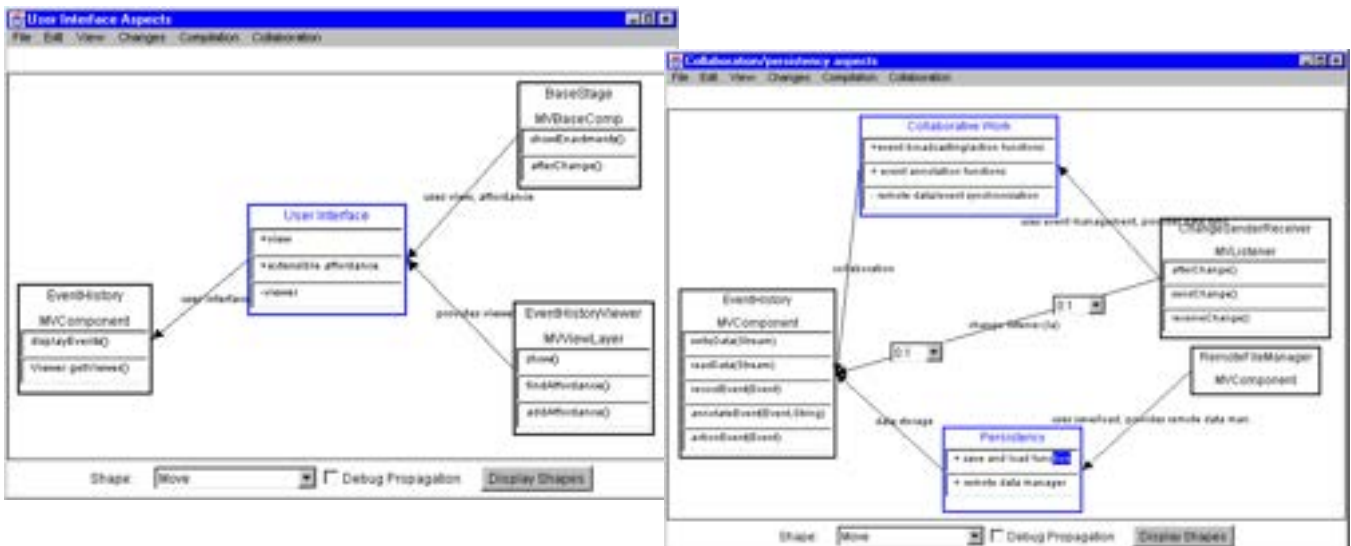


Figure 8. Specifying aspects in JComposer.

8. Summary

Requirements for component-based software systems are difficult to analyse and specify. We have developed an approach that characterises different aspects of a system each component provides to end users or other components, or requires support for from other components. This allows Requirements Engineers to reason about inter-component relationships that exist for a selected component or for a group of related components using categorised perspectives onto a component's data and behaviour. These aspect-oriented views of a component Aspect-oriented component requirements can be refined naturally into design-level software component aspects, and can be encoded into component implementations for use at run-time. We have developed basic tool support for aspect-oriented requirements engineering and used the approach for the reengineering of many components and the development of several new components. The resultant requirements are more easily understood, inter-component relationships reasoned about and component specifications more readily reused than if using traditional requirements engineering approaches.

Acknowledgements

Financial support from the Public Good Science Fund is gratefully acknowledged, as are the many helpful comments of the anonymous reviewers.

References

1. Brown, A.W. and Wallnau, K.C. Engineering of component-based systems, In *Proceedings of the 2nd Int. Conference on Engineering of Complex Computer Systems*, Montreal, Canada, Oct 1996, IEEE CS Press.
2. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, Vol. 2, No. 5, September/October 1998, IEEE CS Press.
3. Grundy, J.C., Mugridge, W.B., Hosking, J.G. Visual specification of multiple view visual environments, In *Proceedings of 1998 IEEE Symposium on Visual Languages*, Halifax, Canada, Sept 1998, IEEE CS Press.
4. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
5. IBM Inc, *VisualAgeTM for Java*, 1998, <http://www.software.ibm.com/ad/vajava/>.
6. IDE Inc., *Software thru PicturesTM 7.0*, 1998, <http://www.ide.com/Products/SMS/core7.0.html>.
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., and Irwin, J. Aspect-oriented Programming, In *Proceedings of the European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241. June 1997.
8. Litva, P.F., Integrating Customer Requirements into Product Designs, *Journal of Product Innovation Management*, Vol. 12, No. 1, pp. 3-15.
9. Netscape Communications Inc, *Visual JavascriptTM*, 1998, <http://www.netscape.com/compprod/products/>.
10. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
11. Park, Y. and Bai, P. Retrieving software components by execution, In *Proceedings of the 1st Component Users Conference*, Munich, July 14-18 1996, SIGS Books.
12. Pressman, R. *Software Engineering : A Practitioner's Approach*, McGraw-Hill, 4th Edition, 1996.
13. Rakotonirainy, A. and Bond, A. A Simple Architecture Description Model, In *Proceedings of TOOLS Pacific'98*, Melbourne, Australia, Nov 24-26, 1998, IEEE CS Press.
14. Rational Corp., *Rational Rose 98*, 1998, <http://www.rational.com/products/rose/prodinfo.html>.
15. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
16. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
17. Szyperski, C.A. and Vernik, R.J. Establishing system-wide properties of component-based systems, *OMG/DARPA Workshop on Compositional Software Architecture*, Monterey CA, Jan 6-8 1998.
18. Wagner, B., Sluijmers, I., Eichelberg, D., and Ackerman, P., Black-box Reuse within Frameworks Based on Visual Programming, In *Proceedings of the 1st Component Users Conference*, Munich, July 14-18 1996, SIGS Books.

3.2 MaramaAIC: Tool Support for Consistency Management and Validation of Requirements

Kamalrudin, M., Grundy, J.C., Hosking, J.G., MaramaAIC: Tool Support for Consistency Management and Validation of Requirements, *Automated Software Engineering*, Springer, 2017, vol 24, no 1, pp. 1-45.
DOI: [10.1007/s10515-016-0192-z](https://doi.org/10.1007/s10515-016-0192-z)

Abstract: Requirements captured by requirements engineers (REs) are commonly inconsistent with their client’s intended requirements and are often error prone. There is limited tool support providing end-to-end support between the REs and their client for the validation and improvement of these requirements. We have developed an automated tool called MaramaAIC (Automated Inconsistency Checker) to address these problems. MaramaAIC provides automated requirements traceability and visual support to identify and highlight inconsistency, incorrectness and incompleteness in captured requirements. MaramaAIC provides an end-to-end rapid prototyping approach together with a patterns library that helps to capture requirements and check the consistency of requirements that have been expressed in textual natural language requirements and then extracted to semi-formal abstract interactions, essential use cases (EUCs) and user interface prototype models. It helps engineers to validate the correctness and completeness of the EUCs modelled requirements by comparing them to “best-practice” templates and generates an abstract prototype in the form of essential user interface prototype models and concrete User Interface views in the form of HTML. We describe its design and implementation together with results of evaluating our tool’s efficacy and performance, and user perception of the tool’s usability and its strengths and weaknesses via a substantial usability study. We also present a qualitative study on the effectiveness of the tool’s end-to-end rapid prototyping approach in improving dialogue between the RE and the client as well as improving the quality of the requirements.

My contribution: Contribution: Co-developed main idea for the approach, co-developed tool design, co-supervised PhD student, wrote substantial parts of paper, co-led investigator for funding for this project from FRST

MaramaAIC: Tool Support for Consistency Management and Validation of Requirements

¹Massila Kamalrudin, ²John Hosking, ³John Grundy

¹*Innovative Software System & Services Group,
Universiti Teknikal Malaysia Melaka, Melaka, Malaysia.*
massila@utem.edu.my

²*Faculty of Science, University of Auckland, Auckland, New Zealand*
j.hosking@auckland.ac.nz

³*School of Software and Electrical Engineering, Swinburne, University of
Technology, PO Box 218, Hawthorn, Victoria 3122, Australia*
jgrundy@swin.edu.au

Abstract: Requirements captured by requirements engineers are commonly inconsistent with their client's intended requirements and are often error prone. There is limited tool support providing end-to-end support between the requirements engineers and their client for the validation and improvement of these requirements. We have developed an automated tool called MaramaAIC (Automated Inconsistency Checker) to address these problems. MaramaAIC provides automated requirements traceability and visual support to identify and highlight inconsistency, incorrectness and incompleteness in captured requirements. MaramaAIC provides an end-to-end rapid prototyping approach together with a patterns library that helps to capture requirements and check the consistency of requirements that have been expressed in textual natural language requirements and then extracted to semi-formal abstract interactions, Essential Use Cases and User Interface prototype models. It helps engineers to validate the correctness and completeness of the Essential Use Case modelled requirements by comparing them to "best-practice" templates and generates an abstract prototype in the form of Essential User Interface (EUI) prototype models and concrete User Interface (UI) views in the form of HTML. We describe its design and implementation together with results of evaluating our tool's efficacy and performance, and user perception of the tool's usability and its strengths and weaknesses via a substantial usability study. We also present a qualitative study on the effectiveness of the tool's end-to-end rapid prototyping approach in improving dialogue between the Requirements Engineer and the client as well as improving the quality of the requirements.

Keyword: *consistency management, requirements validation*

Introduction

A set of requirements is interpreted at the early phase of a system development (Kotonya 1998) and it reflects the client's need for a system. It describes "*how the system should behave, constraints on the system's application domain information, constraints on the system operation or specification of a system property or attribute*" (Kotonya 1998). Software requirement specifications elaborate the functional and non-functional requirements, design artifacts, business processes and other aspects of a software system. Software requirement specifications that are complete and accepted by developers and clients provide a shared understanding and agreement of what a software system should do and why. Since requirement documents form the basis of this agreement and subsequent development processes, they should be correct, complete, and unambiguous (Denger, Berry & Kamsties 2003) and need to be analysed with respect to Consistency, Completeness and Correctness (the "3 Cs") to detect errors such as inconsistency and incompleteness (Biddle, Noble & Tempero 2002).

It is common to find inconsistencies in requirements specifications as the requirements elicitation process involves two or more parties in delivering and understanding correct requirements. Zowghi et al.(2003) assert that expression by different stakeholders may lead to inconsistencies and contradictions because the parties keep changing their minds throughout the development process. *Inconsistent requirements* occur when two or more stakeholders have differing, conflicting requirements and/or the captured requirements from stakeholders are internally

inconsistent when two or more elements overlap and are not aligned (Zisman 2001), (Nuseibeh, Easterbrook & Russo 2000). Typically the relationship is articulated as a consistency rule against which a description can be checked. Inconsistency in requirements also occurs when there are incorrect actions (Fabbrini, Fusani, Gnesi & Lami 2001), or where requirements clash because of disagreements about opinions and bad dependencies (Satyajit, Hrushiksha, & George 2005), sometimes resulting from a lack of skill of different users dealing with shared or related objects. In addition, Litvak (2003) believes that inconsistency occurs when the same parts of the model are portrayed by multiple diagrams and Lamsweerde et al. (1998) find that inconsistency occurs in a set of descriptions when the descriptions can't be satisfied together.

In the context of our research, inconsistencies happen when any of the requirements components that are intended to be equivalent are not. This could be by not being in the same sequence, not having the same name, not being consistent when equivalent components are changed and not being consistent across differing representational models. Positive and negative outcomes for the system development lifecycle are caused by inconsistency (Zisman 2001). Inconsistency highlights contradictory views, perceptions and goals among stakeholders who are involved in a particular development process. It also helps to identify which parts of the system needs further analysis, as well as helping to facilitate the discovery and evocation of the options and information of a system. In addition, Nuseibeh et al. believe that inconsistency can be used as a tool to verify and validate the software process (Nuseibeh et al. 2000). However, it is still vital to avoid or check for inconsistency as it could affect the whole development process, as the clients' requirement needs cannot be met and attempts to do so may cause delay, increase the cost of the system development process, put at risk properties related to the quality of a system and make the maintenance process of a system cumbersome.

Use cases have been used in Requirements Engineering for many years as a semi-formal way to model software requirements from the perspective of user interaction with a system (Jacobson et al. 1999). They are one of the most widely used approaches to semi-formal requirements modelling and a number of research efforts have focused on their capture and analysis. Essential Use Cases, developed by Constantine and Lockwood, offer a simplified, more abstract approach to modelling user/system interaction. They offer advantages of a more abstract and thus simpler form, more structured rules concerning the capture of the EUC models from natural language requirements, and a set of best-practice EUC patterns than can be used for analysis of incorrectness, inconsistency and incompleteness issues (constantine and Lockwood, 1999). Rapid user interface prototypes have been used, often with User Case scenarios, to help refine use case-based requirements by modelling simplified forms of user interfaces to target systems. They offer a way to enhance use case-based requirements especially for stakeholders where a model of the target system interface can be visualized early during requirements capture to give an idea of what it will look like and potentially behave. Essential User Interface prototypes can be used with EUC models to again provide more abstract representations of key target interface components to assist in requirements capture, and for incompleteness and incorrectness analysis.

In order to help engineers to achieve consistency of requirements and to help control and track requirement changes, good tool support is needed (Yufei, Tao, Tianhua & Lin 2010). There are many commercial requirements management tools including DOORS (Hull, Jackson & Dick 2005), Serena RTM (Inc. 2011), Caliber RM (Corporation 1994 - 2010) and Requisite Pro (IBM). They provide good

coverage of requirements management but limited analysis and validation support such as for consistency, correctness and completeness (Geisser 2007). Many research tools exist, most tending to focus on a partial solution for a particular requirements management process, formalism or analysis task. Use case supporting tools are very common, including many commercial tools but also a range of research prototypes that focus on capture from natural language, translation of UC models into other formal models, and UC-based consistency checking. Essential Use Case modelling tools are limited and have very limited consistency checking support. A range of rapid user interface prototyping tools exist for the requirements engineering domain, but few are integrated with UC or EUC models, and few focus on requirements elicitation and consistency enhancement per se.

Given this need for improved requirements capture and analysis, and current lack of adequate tool support, we wanted to address these issues by helping requirements engineers to capture elicited requirements in a semi-formal manner using the Essential Use Case approach. We wanted to leverage this semi-formal model to provide 3Cs checking for the captured EUC-based requirements models. To achieve this we developed novel automated tool support - called MaramaAIC - that uses semi-formal Essential Use Case (EUC) models and Essential User Interface (EUI) prototype models to support consistency management and requirements validation. Both modelling approaches were chosen as they work well in tandem and focus on presenting the abstract, essential requirements models focusing on user/system interaction (Constantine & Lockwood 2003). Our research question derived from this approach is thus “*can automated support for Essential Use Case and Essential User Interface modelling enhance the consistency management and validation of requirements over manual methods?*”. We have developed a prototype tool and compared its performance to manual extraction and validation methods to try and answer this research question.

The remainder of this paper is organized as follows. Section 2 explains the background of the study by defining the basic terms that are used in this paper and Section 3 discusses related work. The automated tool support is discussed in Section 4 and the pattern libraries are discussed in Section 5. Section 6 illustrates an example of the tool’s usage and Section 7 discusses the architecture and the implementation of the tool. The results of the evaluation are discussed in section 8 and the paper concludes with a summary and future work options.

Background

Use Cases (UC)

Use case modelling of requirements has been used for many years in research and practice. The UML popularized their usage for many software development projects (Jacobson et al. 1999). The key concept of the Use Case is a set of related interactions between a target system end user and a part of the system to achieve a task. A use case describes user interactions with the system as a flow of interactions and responses, and may include pre- and post-conditions (when the use case may be used ; pre-existing system or other state information ; and changes to the state post the use case completion); alternative or exception flows; information required by steps in the use case that is exchanged between the user and system; and sometimes other annotations to indicate system behavior in response to interactions. Scenarios are often used with Use Cases to capture particular examples of user/system interaction including example information exchanged. A great advantage of UC-based models is simplicity of concept, understandability by a wide range of stakeholders, usefulness in constructing acceptance and other tests, and semi-formal nature. Disadvantages are lack of a

formalism resulting in limited ability to check for 3Cs problems, lack of agreement on semantics, informality of meaning due to use of natural language and domain-specific terminology, and the large number of use cases that are needed to model even moderately complex systems.

Common alternatives are more formal representations of requirements, such as i^* (Yu, 1997), KAOS (Dardenne et al. 1993), various logics such as TLA and LTL (Lamport, 2002), and visualisations of information structure and flow, such as activity diagrams and sequence diagrams (Jacobson et al. 1999). Advantages of more formal approaches include their ability to be checked by theorem provers and model checks. Advantages of more diagrammatic forms include ability to model larger aspects of systems, more clearly show alternative and other flows, and ability to show related artefacts impacted by user/system interactions. Disadvantages compared to use case modelling include challenges in scaling the models, particularly many formal models, need for mathematical or other logical understanding by model users, and need for good diagrammatic model authoring tools.

Essential Use Cases (EUC)

The EUC approach is defined by its creators, Constantine and Lockwood, as a “*structured narrative, expressed in a language of the application domain and of users, comprising a simplified, generalized, abstract, technology free and independent description of one task or interaction that is complete, meaningful, and well-defined from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction*” (Constantine & Lockwood 1999). An EUC takes the form of a dialogue between the user and the system. The aim is to support better communication between developers and stakeholders via a technology-free model and to assist better requirements capture. This is achieved by allowing only specific detail relevant to the intended design to be captured (Biddle, Noble & Tempero 2002). Compared to a conventional UML use case, an equivalent EUC description is generally shorter and simpler as it only comprises the essential steps (core requirements) of intrinsic user interest (Biddle et al. 2002). It contains user intentions and system responsibilities to document the user/system interaction without the need to describe a user interface in detail. The abstractions used are more focused towards the steps of the use case rather than narrating the use case as a whole.

A set of essential interactions between user and system are organised into an interaction sequence. Consequently, an EUC specifies the sequence of the abstract steps and captures the core part of the requirements (Biddle et al. 2002). Furthermore, the concept of responsibility in EUC aims to identify “*what the system must do to support the use case*” without being concerned about “*how it should be done*” (Biddle et al. 2002). By exploiting the EUC concept of responsibility, a fruitful research area is to focus on the consistency issues between responsibility concepts in requirements and their related designs. This can potentially be used to improve traceability support. EUCs also benefit the development process as they fit a “*problem-oriented rather than solution-oriented*” approach and thus potentially allow the designers and implementers of the user interface to explore more possibilities (Blackwell et al. 2001) They also allow more rapid development: by using EUCs, it is not necessary to design an actual user interface (Biddle et al. 2002).

Figure 1 shows an example of a textual natural language requirement (left hand side) and an example Essential Use Case (right hand side) capturing this requirement (adapted from (Constantine & Lockwood 2001)). On the left is the

textual natural language requirement from which important phrases are extracted (highlighted). From each of these, a specific key phrase (essential requirement) called an abstract interaction is abstracted and is shown in the Essential Use case on the right as user intentions and system responsibilities.

This assists in abstracting the requirements away from for specific technologies. For example, the requirement of typing in login information compared to using biometrics as alternative identification technologies are transformed to a more abstract expression of requirement called “identify self”.

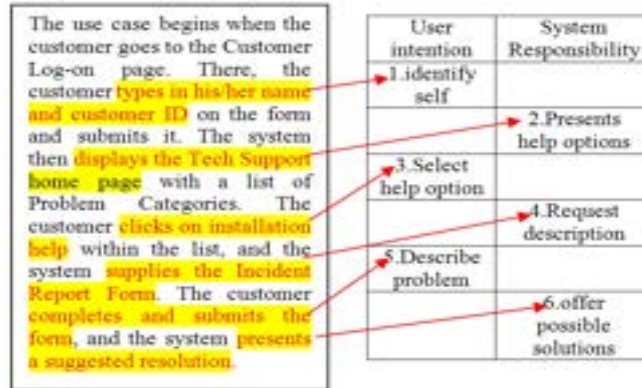


Fig 1. Example of generated EUC model (right) from the textual natural language requirements (left) adapted from (Constantine& Lockwood 2001).

Although EUCs simplify captured requirements compared to conventional UML use cases, requirements engineers still face the problem of correctly defining the level of abstraction which requires effort and time. (Biddle et al., 2000). Requirements engineers need to abstract the essential requirements (using the EUC concept of abstract interactions) manually. This involves understanding the natural language requirements and then extracting an appropriately abstract essential requirement embedded in a logical interaction sequence.

Rapid User Interface Prototyping

Rapid prototyping assists the requirement elicitation process by supporting requirements engineers to gain early feedback from clients on the captured requirements by putting them into a more tangible form i.e. a model of the target system user interface implementing those requirements (Robertson 2006), (Buskirk & Moroney 2003). Low-fidelity or abstract prototypes (often paper) are commonly used in this process (Constantine 1998). The idea is to make the captured requirements for e.g. a use case much more tangible to the stakeholder by giving them a semblance of the target system if it were implemented based on these requirements. The stakeholder is then able to more concretely understand how the requirements might be realized and if the captured requirements are indeed consistent, complete and correct. A rapid User Interface (UI) prototyping approach can thus complement other capture and checking approaches used.

Types of abstract prototypes include abstract user interfaces (Cristian 2008), UI prototypes (Memmel & Reiterer 2009) and EUI prototypes (Constantine & Lockwood 1999). These are all easy-to-change mock ups which encourage iteration of the elicitation and validation process (Robertson 2006), (Memmel & Reiterer 2009). They allow a rough walk-through of user tasks before needing to factor in hardware or technology concerns (Buskirk & Moroney 2003) and can avoid clients being fixated at an early stage on concrete product appearance rather than functionality (Robertson 2006). However, previous work has shown that the application of low-fidelity techniques in practice can prove challenging (Robertson

2006), due to lack of tool support and lack of integration between models, processes and analysis support.

An example of rapid prototyping is Essential User Interface (EUI) prototyping, a low-fidelity prototyping approach (Ambler 2003-2009). It provides the general idea behind the UI but not its exact details. It focuses on the requirements and not the design, representing UI requirements without the need for prototyping tools or widgets to draw the UI (Constantine & Lockwood 2003). EUI prototyping extends from, and works in tandem with, the semi-formal representation of EUCs, both focusing on users and their usage of the system, rather than on system features (Ambler 2004). It thus helps to avoid clients and REs being misled or confused by chaotic, rapidly evolving and distracting details. Being primarily a whiteboard or paper-based technique to date, it does not integrate well with most other tools used in the software engineering process (Ambler 2003-2009). However, it shows promise as a way to complement EUC-based semi-formal models by surfacing the requirements using an abstract user interface model.

Figure 2, from Ambler (2004), shows an example of an EUI prototype being developed from an Essential Use Case (EUC). The post-it notes represent abstractions of user interfaces. The different colours of these notes represent different UI elements. Pink notes represent the input field, yellow notes represent display only and blue notes represent actions (Ambler 2004). Here, the Requirements Engineer (RE) is capturing the user intention/system responsibility dialogue represented in the EUC as possible UI functionality at a high level of abstraction.

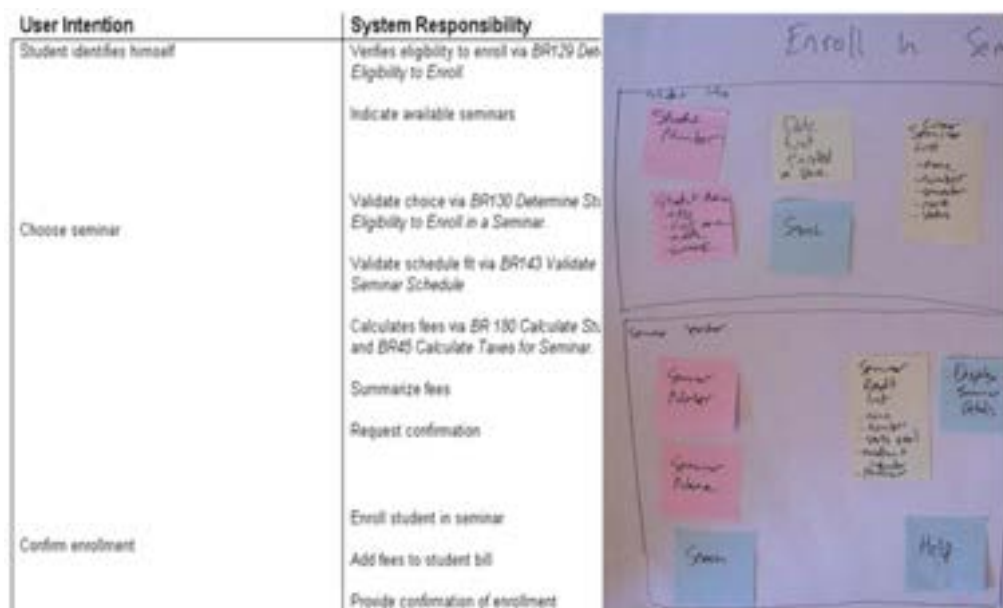


Fig 2. Example of EUI prototype iterates from Essential Use Cases ((Ambler 2004)), (Hull, Jackson, & Dick 2005))

Related Work

Much research has been devoted to developing tools for managing the consistency of or checking for inconsistency in requirements using formal or semi-formal specifications. For example, E-Lopez-Herrejon and Egyed (2012) presented work on fixing inconsistencies with variability by using and comparing two approaches: random and heuristic, based on the size of the fixing set and the time taken for the execution by having a DFS-based approach as a baseline. However, further consideration is needed for both fixing multiple inconsistency rule instances

and having more complex fixing operations (Lopez-Herrejon & Egyed 2012). Other work by Reder and Egyed (2012) provides an automated tool for incrementally validating design rules in a validation tree to improve the performance of incremental consistency checking. Their automated tool support was found to be able to minimise the time taken for re-validation of design rule and fits well with all kind of design rules. However, it is only focussed on validating parts that are affected by model changes and not all design rules (Reder & Egyed 2012). Egyed (2001) has also implemented a UML-based transformation framework to check inconsistency and help in comparison using an automated checking tool called VIEWINTEGRA. This uses consistent transformation to translate diagrams into interpretations and used the consistency comparison to compare those interpretations with those of other diagrams (Egyed 2001). This technique can check inconsistencies without the help of third party or intermediate languages. The limitation of this tool exists when checking the consistency between an object diagram and state chart diagram or vice versa, as they cannot be transformed directly and need to be changed to a class diagram first in order to obtain consistency results (Egyed 2001).

Another approach is presented by Perrouin et al. (2009) for managing the inconsistencies amongst heterogeneous models by using a model composition mechanism. The information from the heterogeneous models is translated to a set of model fragments (Perrouin, Brottier, Baudry & Le Traon 2009). Fusion is applied to build a global model which allows various inconsistencies to be detected, resulting in the global model (Perrouin, Brottier, Baudry & Le Traon 2009). Automation is applied to compute traceability links between the input model and the global one and thus support the reporting of the inconsistencies on the original model and help to resolve the cause of those inconsistencies (Perrouin, Brottier, Baudry & Le Traon 2009). However, a classification of which inconsistencies need to be resolved is not provided (Perrouin, Brottier, Baudry & Le Traon).

Nentwich et al. (2003) proposed a repair framework for inconsistent, distributed documents (Nentwich, Wolfgang & Anthony, 2003). They generate interactive repairs from a first order logic formula that constrains the documents. Their repair system provides a correct repair action for each inconsistency together with available choices. However, they face problems when the repair actions interact with the grammar in a document, and also actions generated by other constraints (Nentwich, Emmerich, Finkelstein and Ellmer 2003). Their approach also fails to identify a single inconsistency that may lead to other inconsistencies (Nentwich, Wolfgang & Anthony 2003). Gervasi and Zowghi (2005) used the tool in detecting, analysing and handling inconsistencies in requirements for various stakeholders. This work extended the tool to employ theorem proving and model checking in the context of default logic to deal with the problems in a formal manner. The tool's limitation is that propositional logic used is not powerful enough to model complex system behaviour (Gervasi & Zowghi 2005).

There has also been work done to check the consistency of aspect-oriented requirements. Sardinha et al. (2012) developed an automated conflict detector called Early Aspect Analyzer (EA-Analyzer) based on a Bayesian learning method for a large set of aspect-oriented requirements compositions. This tool demonstrates the benefits of Aspect Oriented Requirements Engineering (AORE) to detect and to analyse conflicts in the AO requirements text, but it requires training before using the tool and needs wider requirements sets to test the scalability of the tool. The tool also does not operate alone as it requires assistance from the EA-Miner tool which is developed by Sampaio et al. (2005) to identify and separate concerns,

either aspectual or non-aspectual (Sampaio et al. 2005). Other work relating to checking consistency using an aspect-oriented paradigm, this time for web applications, is by Yijun (2004). The author presents a tool called HILA which was designed as an extension of UML state machines to model the adaptation rules for web applications (Yijun 2004). However, this work is not limited to web engineering applications but may also be applicable to other areas (Yijun 2004). HILA could be helpful in improving the modularity of models and helps to automate the consistency checking of aspects to ensure rules are always in a consistent state (Yijun 2004). Likewise, Zhang and Holzl (2012) uses HILA with their weaving algorithm and implementation of semantic aspects to check and to resolve conflicts between various aspects. Then Zhang (2012) also uses HILA to model the mutual exclusion requirements in a specified place. This work is also found could minimises potential conflicts between aspects. On the other hand, Yue et al. (2015) developed a method and tool called aToucan to automatically transform use case model to analysis models such as class, sequence and activity diagrams. Here, the traceability is established between both the use case and analysis model where it help to maintain the models when changes happen and somehow indirectly help to ensure complete, correct and consistent UML model comprising of both structural and behavioral aspects via an intermediate model to be generated.

Nguyen et al. (2012) developed an automated tool called REInDetector a knowledge-based engineering tool to capture and to detect a range of inconsistencies of requirements. This tool uses descriptive logic (DL) as its formal basis of object/class-style ontologies to formalise and analyse requirements (Nguyen, Vo, Lumpe & Grundy 2012). The tool can identify missing elements and conflicts in requirements (Nguyen et al. 2012). However, there is very limited support for temporal operators in DL and this does not allow the tool to detect conflicts associated with the requirements that are not expressible in the DL.

To summarize, many techniques discussed above are reasonably well developed and evaluated but most are immature. Most work uses tool support for the checking process. However, most of these integrate with other available tools and are not purely built for consistency checking, especially when this needs to deal with processing natural language or to formalize the requirements. Most tools or approaches lack rigorous checking for consistency as they only support partial solutions for checking or identifying inconsistency and with a homogeneous model of a set of requirements. We also identified that the tools developed need human intervention to interpret the consistency results or invoke actions to check for inconsistency. Semi-formal specifications are of great interest although some studies concluded that maintaining consistency between models is not important and expensive (Kovacevic 1999). Almost no research has been undertaken on managing consistency using the Essential Use Case representation (Biddle April 2000). Very little of the identified research work provides tools to handle full end-to-end consistency checking support, i.e from the natural language requirement to models and then to a user interface prototype. Most work is only concerned with validating requirements by requirements engineers and not by the clients.

Preliminary Experience in Applying EUCs and EUIs

Previous research using the EUC approach to model software requirements has indicated that requirements engineers sometimes have difficulty in identifying the “abstract interactions” used by EUCs and their sequencing (Biddle.R April 2000). To obtain a better understanding of these potential difficulties, we conducted a user study with 11 post-graduate software engineering students, several of whom

had previously worked in industry as developers and/or requirements engineers. All were very familiar with UML use case modeling and most had used UML use cases to model requirements previously. None were familiar with the EUC modeling approach. This allowed us to see ways in which novice EUC users could be supported by a tool. Though we used students and inexperienced requirements engineers without much EUC experience, it does not impact our study results as we wanted to precisely understand the challenges faced by such novice EUC users.

The participants carried out the extraction of an EUC model from a set of requirements specified in natural language, in order to observe their performance and understand their experiences in using EUCs. We used the same sets of requirements for modelling (Constantine & Lockwood 2001) and compared the EUC models developed by EUC novices with the ones produced by a modeler familiar with EUCs (Kamalrudin 2010) (Kamalrudin & Grundy 2011).

In this study the average time taken to accomplish the EUC development task was 11.2 minutes. The longest time taken was about 25 minutes and the shortest time taken was about 5 minutes, so there was significant variation in the time taken. Also participants were more likely to generate incorrect EUC interactions than correct ones, and very unlikely (9.1%) to produce a completely correct EUC. All but one participant failed to identify some of the essential interactions present in the natural language requirements; many failed to assemble these into an appropriate interaction sequence. The root cause of most problems was that participants tended to incorrectly determine the required level of abstraction for their essential interactions (the user intentions and system responsibilities of the EUC model). This is based on observations made as they performed the task as well as analysis of the answers provided by them. The study also demonstrates that it was quite time consuming for participants as they needed to figure out the appropriate keywords that describe each abstract interaction and to organise them into an appropriate sequence of user intentions and system responsibilities.

We then conducted a similar study with the same scenario to understand further the problem faced by requirements engineer in applying the EUI prototype model approach. This second study involved 20 post-graduate software engineering students, several of whom had previously worked in the industry as developers and/or Requirements engineers. All were familiar with requirements and prototyping at the elicitation phase, but none with the EUI prototyping approach. Each participant was given a brief tutorial on the approach and examples of natural language requirements with derived EUC models and EUI prototypes. Participants were then asked to develop an EUI prototype model from an EUC model and natural language requirements. Here, we also tracked the time they took to complete their tasks.

As with the EUC study, participants were found to be more likely to generate incorrect EUI prototype models than correct ones. This is because the participants tended to incorrectly determine the main UI component of a specific business use case. Almost all participants tended to capture unnecessary UI components, gearing towards a concrete GUI rather than EUI components. There was also considerable variation in the time taken and the longest time taken did not increase the likelihood of the correctness of the answer. Our studies thus support the anecdotal findings reported in (Biddle 2000) regarding the problems faced in extracting the correct abstract interaction of EUCs and using low-fidelity prototypes but with more quantitative evidence.

Automated Tool Support: MaramaAIC

The results of these preliminary studies motivated us to develop new automated tool support to enable requirements engineers to effectively capture or confirm more requirements with clients at an early stage of requirement analysis. We wanted to support an end-to-end rapid prototyping approach which uses low-fidelity EUI prototyping together with a concrete UI prototype. Our new tool, MaramaAIC (Automated Inconsistency Checker) provides a range of inconsistency checking which is not limited to a partial solution or partial components to be checked. Figure 3 shows the way MaramaAIC is used.

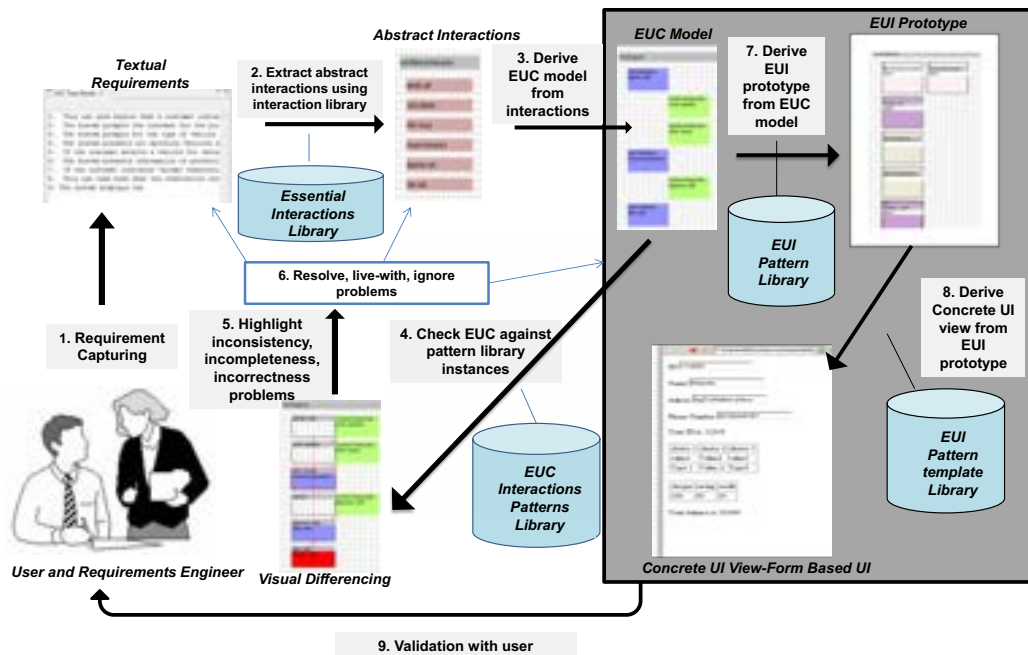


Fig 3. Usage of Marama AIC

Marama AIC improves traceability by implementing a lightweight approach together with a traceability technique and semi-formal specification in the form of EUC models in order to support consistency checking between the natural language requirement, an EUC model and an EUI prototype.

In Step 1, Requirements are first captured from natural language text.

In Step 2, Abstract interactions are then extracted using lightweight the natural language processing of phrase and regular expressions based on a essential interaction library, following the EUC approach described by Constantine and Lockwood and refined by Biddle et al.

In Step 3, an EUC model is then generated from the abstract interactions.

In Step 4, Extracted EUC models are checked against a repository of best practice EUC models derived from Biddle et al's work and our own. Sequences of EUC interactions can be compared to common sequences, or EUC interaction patterns, in our EUC interaction patterns library using this "differencing" concept.

In Step 5, Visual highlighting is used to warn the user of inconsistencies in any requirements element.

In Step 6, the requirements engineer can choose to resolve inconsistency, incompleteness and/or incorrectness problems detected, leave highlighted problem markers and later resolve them, or ignore problems until later.

In Step7, the tool also allows the requirements engineer to automatically and traceably transform EUC models to EUI prototypes using our novel EUI pattern

library. This means traceability is provided throughout the process, allowing any of the EUI components to be traced forward/back from/to the EUC model, abstract interaction or textual natural language requirement.

In Step 8, MaramaAIC allows the EUI prototype to be translated to a more concrete form-based UI view, an HTML form, by using a novel EUI Pattern template library. An EUI prototype model can also be translated to a concrete form-based UI using a pre-defined template in a EUI pattern template library, with one template for each EUI pattern. Here, the EUI Pattern template consists of the descriptions of Concrete UI components to be instantiated for a particular EUI pattern. Simple interaction with the generated HTML form is also supported to illustrate how target system information input and output could work.

In Step 9, the EUI model and concrete UI generated from the tool can be reviewed by the requirements engineers with end-users to validate and confirm the consistency of the original textual requirements.

To achieve Steps 4 and 5 the extracted EUC model's abstract interactions are compared to an expected essential interaction and EUC pattern's set of abstract interactions and their sequencing. When any problems with requirements models are detected, the tool focuses on providing warning, feedback notification and visualisation of the quality issues existing in any component:

- Components that mismatch, do not exist in one model, have differing sequencing between components, or that overlap with non-corresponding names or other information, are classed as an “inconsistency”.
- Detected redundancy of a component or a mismatch between a component and the expected element in an otherwise matching pattern is classed as “incorrectness”.
- Missing components or sequences in a model compared to an otherwise matching pattern are classed as “incomplete”. The set of requirements is assumed to be “complete” (Huzar, Kuzniarz, Reggio, & Sourrouille 2005) once all the requirements model elements satisfy a match or matches in the EUC interaction pattern library.

In Step 6, when any of the above problems are highlighted, requirements engineers then have the ability to choose to do one of the following:

- I. Resolve a detected quality issue by modifying the components based on the results of the consistency engine recommendation.
- II. Tolerate the inconsistency until later, with our tool tracking it.
- III. Strictly ignore the inconsistency.

MaramaAIC avoids forcing requirements consistency immediately as consistency rules cannot always automatically maintain the consistency of the set of requirement components. For example, if the sequence of components of the abstract interaction or EUC is problematic, we cannot automatically enforce a change in the structure of the textual natural language as this requires manual intervention. In this situation, a warning and notational element highlighting make users aware that the inconsistency is present. Explicitly ignoring the inconsistency (suppressing warnings) is also allowed as it respects requirements engineers to make the final decision on the quality of their requirements. End-user stakeholders can view updated and/or annotated textual requirements at any time to understand the correctness and completeness of the requirements model. While the EUC model is arguably end-user-friendly, keeping it consistent with the textual natural language

representation affords the latter human-centric views continued use through the requirements engineering process.

EUC and EUI Patterns Libraries

In order to simplify the above EUC and EUI extraction process, we adopted a domain-specific approach, instead of using conventional NLP-based approaches to capture requirements. This means we chose to develop a library of “proven” essential interactions expressed as textual phrases, phrase variants and limited regular expressions. We also developed a library of EUC patterns for higher level consistency checking and an EUI pattern library for the generation of the EUI prototype model (Kamalrudin, Grundy and Hosking 2011).

These libraries of essential interactions, EUC and EUI patterns were developed from a collection of such patterns previously identified by Constantine and Lockwood (1999) and Biddle et al. (2000) together with patterns that were developed by us, which are all applicable across various domains.

Essential Interactions Patterns

We developed an essential interaction pattern library for storing essential interactions and abstract interactions. This essential interaction pattern library is based on a collection of phrases that illustrate the function or behaviour of a system. The collection of phrases is then categorised, based on its related or associated abstract interaction. We have collected and categorised phrases from a wide variety of textual natural language requirements documents available to us and stored them as essential interactions. Currently, we have collected over 360 phrases from various requirement domains including online booking, online banking, mobile systems related to making and receiving calls, online election systems, online business, online registration and e-commerce. The collection and categorisation of the phrases is an on-going process. Based on these phrases, we have come up with close to 80 patterns of abstract interaction. On average, there are 4.5 phrases or essential interactions associated with each abstract interaction. For example the abstract interaction “display error” is associated with four different essential interactions: “display time out”, “show error”, “display error message” and “show problem list”. The essential interactions were not categorized based on one scenario. They have associations with up to five different concrete scenarios such as online business, e-commerce, online booking, online banking and online voting systems. One particular abstract interaction can be thus associated with multiple concrete scenarios. Table 1 shows some other examples of abstract interactions and their associated essential interactions for various domains of application.

Table 1 Example of Abstract Interactions and their Associated Essential Interaction and Their Related Domains

Abstract interaction	Essential interaction	Example of Domains
Verify user	verify customer credential	Online banking, online booking, online business, e-commerce, online reservation
	verify customer id	Online banking, online booking, online business, e-commerce, online reservation
	verify username	Online banking, online booking, online business, e-commerce, online voting system, online reservation
Ask help	help desk	Online banking, online booking, online business, e-commerce, online reservation

	request for help	Online banking, online booking, online business, e-commerce, online voting system, mobile system, online reservation
	ask for help	Online banking, online booking, online business, e-commerce, online voting system, online reservation
	clicks help	Online banking, online booking, online business, e-commerce, online voting system, online reservation
	complete help form	Online banking, online booking, online business, e-commerce, online voting system, online reservation
Offer choice	prompt for amount	Online booking, online banking, online business, e-commerce
	display account menu	Online banking
	display transaction menu	Online banking

In order to store the essential interactions in the essential interaction pattern library, selected phrases (“key textual structures”) are extracted from the textual natural language requirement, based on their sentence structure. The ‘key textual structure’ uses Verb-Phrases (VP) and Noun-Phrases (NP) in the sentence structures to categorise the essential interactions. Any phrases that follow this structure will be acceptable as an essential interaction in the essential interaction pattern library. The tree structure of the key textual structure is illustrated in Figure 4. This shows that our library has three different sentence structures, based on the location of the Verb Phrase (VP) and Noun Phrase (NP). The Noun Phrase can contain structure elements such as Articles (ART) and Adjectives (ADJ) or only Nouns (Noun).

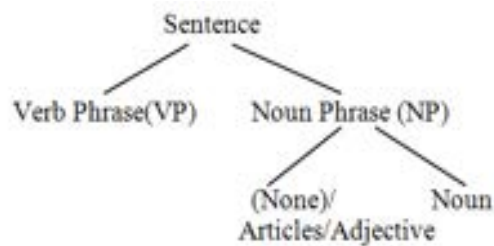


Fig 4. Tree Structure for Key Textual Phrase

The three different sentence structures are:

- I. **Verb (V) + Noun (N) (only)** e.g. request (V) amount (N)
- II. **Verb (V) + Articles (ART)+ Noun (N)** e.g. issue (V) a (ART) receipt (N)
- III. **Verb (V) + Adjective (ADJ)+ Noun (N)** e.g. ask (V) which (ADJ) operation (N)

Below is an example of a part of textual natural language requirements by (Evan, 2009) that comprises the sentence structures of the key textual phrase to store the essential interaction.

“The system **prompts the customer** for the pickup and returns locations of the reservation, as well as the pickup and return dates and times. The customer **indicates the desired locations and dates.** “

It is shown from the example that both underlined and bold requirements follow the key textual phrase of “Verb (VP)-Noun (NP)” but with different location of the Verb Phrase (VP) and Noun Phrase (NP). Both sentences of “prompts the customer” and “indicates the desired locations and dates” follow the second structure (II) of Verb+Article+Noun.

This key textual structure aims to provide flexibility in the library’s ability to accommodate various types of sentences containing abstract interactions. With this, a broad range of phrase options can be extracted by the tracing engine, while

still affording a lightweight implementation using string manipulation and some regular expression matching.

EUC Interaction Patterns

A set of best practice EUC interaction patterns or templates was developed based on a range of typical user/system interactions in a wide variety of domains (Biddle April 2000). The EUC interaction patterns library stores these best practice patterns of EUCs for each set of scenarios or use case stories. Table 2 illustrates some examples of EUC interaction patterns for scenarios such as “reserve item” and “purchase item”, with their sequences of abstract interactions. We use these “best-practice” templates for higher level checking of consistency, correctness and completeness of a generated EUC model by comparing the EUCs to the templates.

Table 2 Examples of EUC Interaction Patterns

Scenarios/ Use Case stories	User intention Abstract Interaction	System responsibility Abstract Interaction
Reserve item	Choose	
		offer choice
		view detail
		request identification
	identify self	
Purchase item		confirm booking
	Choose	
		check status
	identify self	
	provides detail	
		verify identity
	request confirmation	
	view detail	

EUI Patterns and EUI Pattern Templates

We also developed a set of EUI patterns in an EUI Pattern library, using an adaptation of the brainstorming methodology proposed by Constantine and Lockwood (1999). This adaptation generalised their approach by providing a simpler and more generic EUI pattern for EUI prototypes. The generalised EUI pattern comprises four types of EUI pattern category: List, Display, Input and Action. These are similar to the concept of Containers, introduced by Constantine and Lockwood. The main aim of these EUI Patterns is to assist REs to rapidly model a user interface based on the requirements captured and modelled earlier in the EUC model. An abstract UI captured using such a pattern is used as a medium for early communication between the RE and the client as it is easy to understand and allows the client to narrow down UI detail before moving to the concrete UI. In more detail, the four EUI pattern categories are as follows.

- **List:** Show a list of items, options or values that are associated with a particular abstract interaction of the EUC model. Default values are provided from the UI pattern library but can be overridden during application.
- **Display:** Display output based on an associated abstract interaction of the EUC model. This could display a name, id, number, address, message or notification.

- **Input:** Allow a user to input data or details of a specific element associated with an abstract EUC interaction.
- **Action:** Show a control button, such as save, delete and submit, based on an associated EUC abstract interaction.

Each of these EUI patterns is associated with an abstract interaction from the EUCs. An EUI pattern can be associated with one or multiple abstract interactions. Table 3 shows some examples of mappings between abstract EUC interactions (right) and various EUI patterns (centre), and their categories (left). For example, the EUI pattern “Save” from the “Action” category is associated with three different abstract EUC interactions: “record call”, “record detail” and “save identification”. We can see that the abstract EUI patterns are very general and apply across a range of different domains. For example, the EUI pattern “Save” could support a range of different scenario domains such as making calls in a mobile application domain to online booking, registration and retail systems.

Table 3 Example of EUI pattern Category and its related EUI pattern and it’s associated Abstract Interaction from the EUC model

EUI pattern category	EUI pattern	Abstract interaction
List	List of option	Choose
		offer choice
		Select option
	List of solution	offer alternative
		offer possible solution
	List of payment	choose transaction
		choose payment
select amount		
Display	Display payment	validate payment
	Display Item detail	show payment
		return item
	Display status	view detail
	Display ID	check user
		Notify user
Display error message	verify identity	
Input	ID	provide identification
		display error
	Other personal detail	identify self
		request identification
	Payment detail	identify self
	Item detail	request identification
	Number	make payment
provides detail		
Action	Help	make call
		indicates number to dial
	Save	Ask help
		Present solution
		record call
	Print	Record detail
		save identification
Delete	Print	
	delete item	

The EUI Pattern template library is comprised of EUI Pattern templates which support translating the EUI prototype to concrete UIs in a form of HTML pages. An EUI pattern template is based on the EUI pattern used in the EUI prototype. The EUI pattern template is already pre-defined in the library. It contains templates defined in HTML format for each of the EUI pattern categories: List, Display Input and Action. The defined EUI Pattern template for the HTML form is as below;

- i. List: Table
- ii. Display: message/text/data/value
- iii. Input: Text Input
- iv. Action: Button

The EUI pattern template is also applicable and reusable for various domains of applications. Table 4 shows examples of EUI pattern templates with their associated EUI patterns and domains applicable to the pattern.

Table 4 Examples of EUI Pattern template with its associated EUI Pattern and associated Domains in the EUI Pattern template library

EUI pattern categories	EUI Pattern	EUI Pattern template	Domains
Action	Submit	Button	Online banking, online booking, online business, e-commerce, online voting system, mobile system, online reservation
	Add		
	Search		
List	List of item	Table	Online banking, online booking, online business, e-commerce, online voting system, mobile system, online reservation
	List of payment		
	List of option		
Display	Display availability	Numbers/text	Online banking, online booking, online business, e-commerce, online voting system, mobile system, online reservation
	Display amount	Value/text	
	Display ID	Numbers	
Input	Item detail	Text input	Online banking, online booking, online business, e-commerce, online voting system, online reservation
	Payment detail		
	Problem form		

1. This use case begins when a customer indicates he wishes to make a reservation for a rental car.
2. The system prompts the customer for the pickup and returns locations of the reservation, as well as the pickup and return dates and times. The customer indicates the desired locations and dates.
3. The system prompts for the type of vehicle the customer desires. The customer indicates the vehicle type.
4. The system presents all matching vehicles available at the pickup location for the selected date and time. If the customer requests detailed information on a particular vehicle, the system presents this information to the customer.
5. If the customer selects a vehicle for rental, the system prompts for information identifying the customer (full name, telephone number, email address for confirmation, etc.). The customer provides the required information.
6. The system presents information on protection products (such as damage waiver, personal accident insurance) and asks the customer to accept or decline each product. The customer indicates his choices.
7. If the customer indicates "accept reservation," the system informs the customer that the reservation has been completed, and presents the customer a reservation confirmation.
8. This use case ends when the reservation confirmation has been presented to the customer.

Fig 5. Example of User Scenario: Reserve a Vehicle (Evans, 2009)

Tool Usage Example

In this section we illustrate the use of MaramaAIC using requirements which was developed by Evans and published on the IBM developer works website, as an example of a requirement to demonstrate the key features of our tool. This user scenario is a “hypothetical browser-based software system for an auto rental company” (Evans, 2009) mainly for an individual account. It illustrates the situation that happens in a rental company when a customer comes to the rental counter to

rent a vehicle (Evans, 2009). It is also an example from an online booking domain of application. The description of this user scenario is shown in Figure 5.

Example of Usage

Nancy, a requirement engineer, would like to validate the requirements that she has collected from the client, John, who is the car rental information manager. To do this, as shown in Figure 6, she types in the requirements in a form of user scenario to the textual editor or copies them in from an existing file (1) and has the tool trace the essential requirements (abstract interactions) (2). Here, she verifies the list of abstract interactions provided by the tool and then has the tool generate the EUC model (3). In order to check for the consistency and dependencies among the EUC component and the abstract interaction and the user scenario, she performs trace back by using the event handler from the EUC component or abstract interaction. For trace back (as shown in Figure 6), the selected EUC component (A) and its associated abstract interaction (B) changes colour to red and the associated essential interactions (C) are highlighted with “***”. The processes of tracing forward/backward and mapping are assisted by event handlers. These tracings show and maintain the consistency among the requirement components.

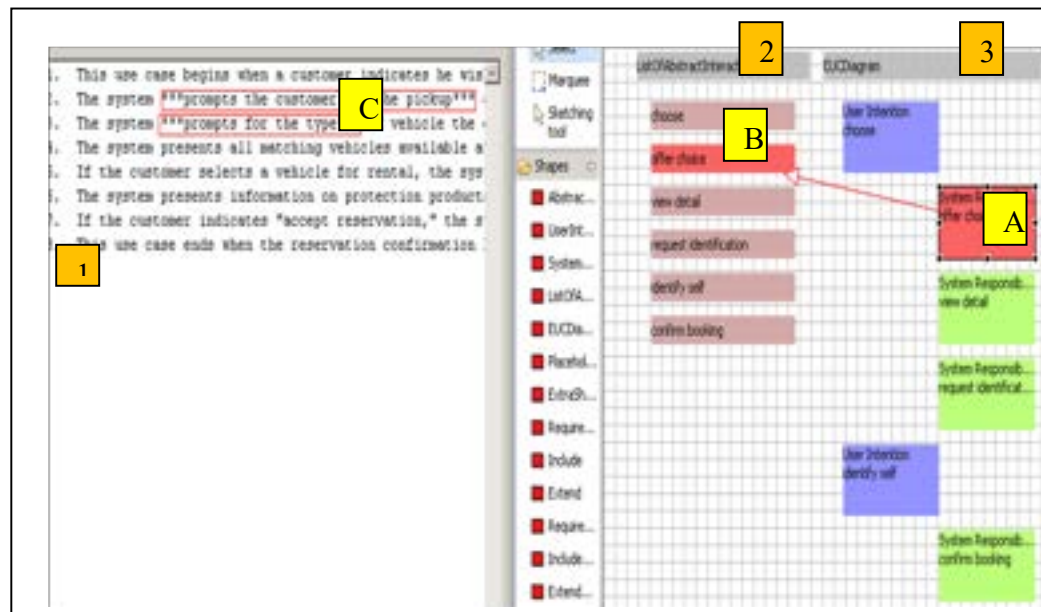


Fig 6. Capturing requirements - trace the abstract interaction, trace back and map to EUC model

By using MaramaAIC, Nancy can make any modification to any of the requirement components if she is not satisfied with the results provided by the tool. For example, if she thinks one of the abstract interactions is missing, she could add a new abstract interaction to the list. In particular, she might think that an abstract interaction “make payment” is missing from the list. Thus, she adds a new abstract interaction “make payment” to the list. This action triggers an inconsistency warning and the options either to update, delete or continue without updating the textual natural language requirements to appear to inform her that an inconsistency has occurred in the requirement components (as shown in Figure 7 (1)). She then

chooses to continue without updating the user scenario as she probably thinks that the “make payment” abstract interaction is necessary and matches the user scenario. Although the option “continue” is chosen by her, she can still map the newly-added abstract interaction to the EUC model (2). This triggers a problem marker to inform her of the inconsistency error for later consideration to resolve the inconsistency (3).

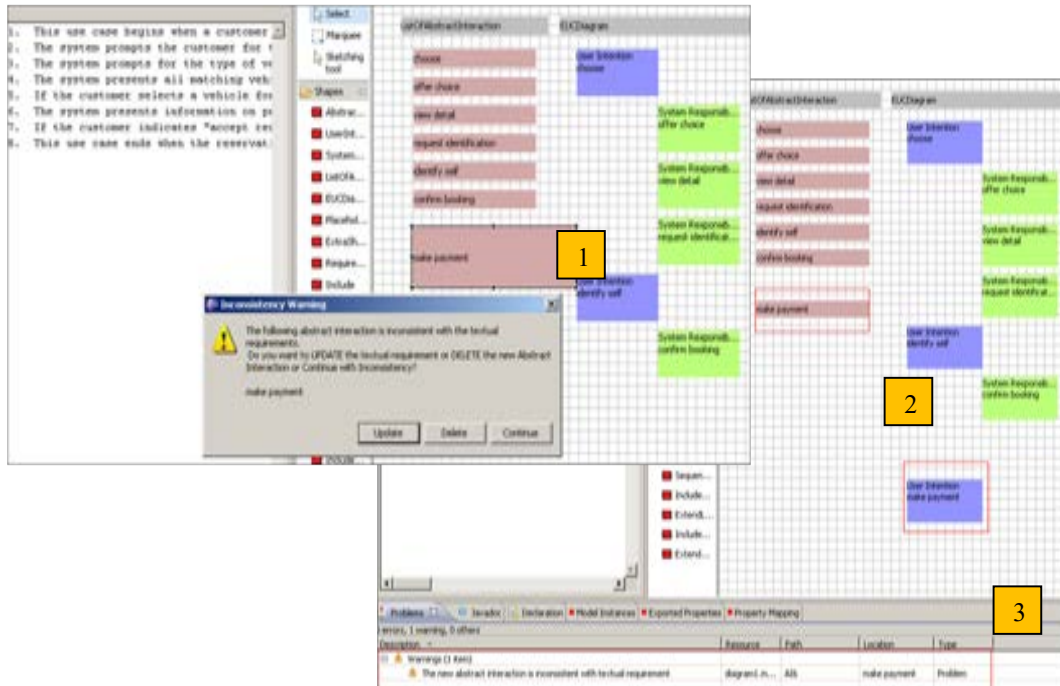


Fig 7. Add New Item to Abstract Interaction

NEXT, Nancy is also unhappy with the sequence ordering of one the abstract interaction components: “choose”. She thinks this abstract interaction should be above the “make payment” component as shown in Figure 8 (1) because the user should choose from the option before any payment should be requested. This triggers the associated EUC component “choose” to change colour to red and the essential interaction “indicates” to be highlighted with “***”. An Inconsistency warning also appears to inform her of the inconsistencies and provide options either to update or cancel the change. A problem marker also provides warning on inconsistencies that still exist. Then she decides to update the sequence ordering, and this automatically also changes the position of the EUC component “choose” (2). However, the ordering of the highlighted essential interactions is not altered as such changes could affect the structure of the user scenario. This action also triggers a problem marker to warn about the inconsistencies that have not been completely resolved.



Fig 8. Change of Abstract Interaction Sequence Ordering

On reviewing the extracted EUC, Nancy feels that there is an extra component in the EUC model. She thinks that the EUC component “offer choice” is not necessary and needs to be deleted. She believes there is a redundancy between the “choose” and “offer choice” component as shown in Figure 9 (1). Thus, she selects the “offer choice” component to be deleted. This action triggers the associated abstract interaction to automatically change colour to red and the associated essential interactions “prompts the customer for the pickup” and “prompts for the type” to be highlighted with “***” as shown in Figure 9 (2). The inconsistency warning also appears to inform the inconsistencies and options to either delete or cancel the deletion. Although a notification of the inconsistencies is provided, she still thinks she needs to delete the “offer choice” component. This triggers the associated abstract interaction and essential interactions also to be deleted. This occurs as the tool tries to keep all the three requirement components in a consistent state.

Being a novice requirement engineer, Nancy is keen to validate her extracted EUC model against a best-practice EUC template. Thus, she looks through the list of available templates and chooses the pattern “Reserve Item” as shown in Figure 10 (1) that appears to be similar to this scenario. She matches the pattern to her EUC model and sees that she has missed some interactions as a few sequence orderings and components are incorrect. In addition, an extra component also exists in the interaction. As shown in Figure 10 (2), the incorrect sequence ordering is shown by the red visual links (A), the existence of the extra component “make payment” (B) is outlined with red and the correct component “offer choice” (C) is shown by a grey element on top of the green shape “view detail” which also displays the incorrect component and position held by the “view detail” component. As there is an unmatched interaction between the generated EUC and the best-practice template, Nancy is notified with an inconsistency warning and given options to either keep or change the generated EUC following the best-practice template. She agrees with the warning and the errors shown. She then selects to change this EUC model to the EUC interaction templates.

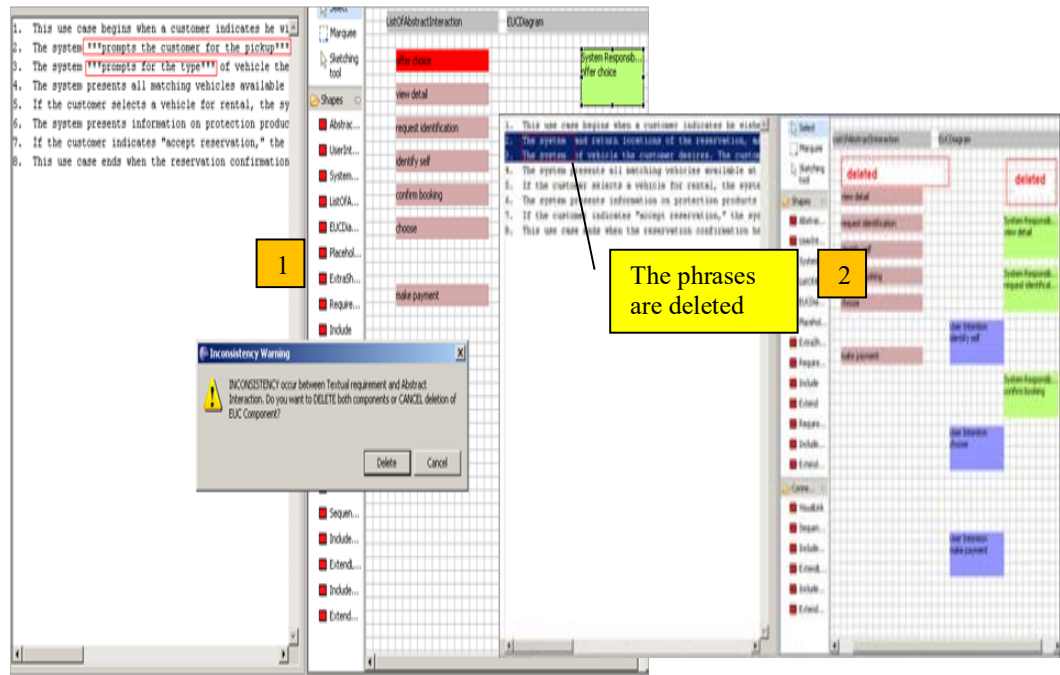


Fig 9. Delete the EUC component.

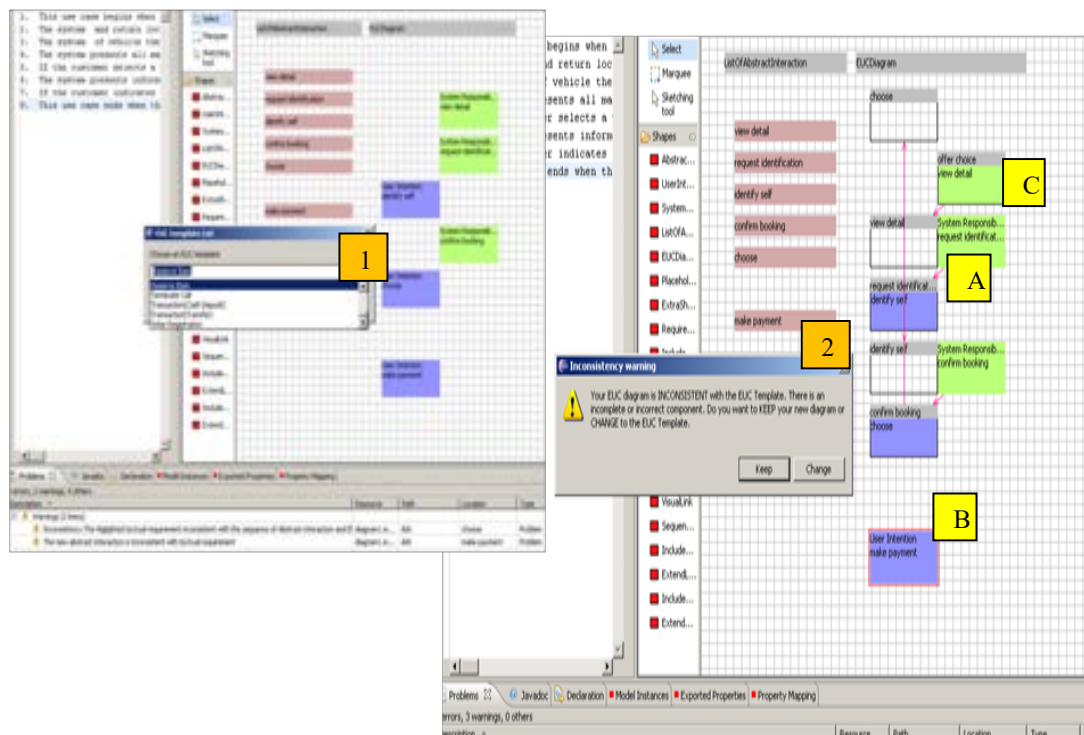


Fig 10. Visual differencing to check for incorrectness and incompleteness

When Nancy is satisfied with the requirements components, she sits with John to validate the requirements and to confirm the consistency of her captured requirements with the earlier requirements provided by John. In order to allow John to better understand the requirement components, she then has the tool map the EUC model to abstract prototype: EUI prototype as (1) and also has the tool translate EUI prototype to a concrete UI view in a HTML form (2) as shown in Figure 11.

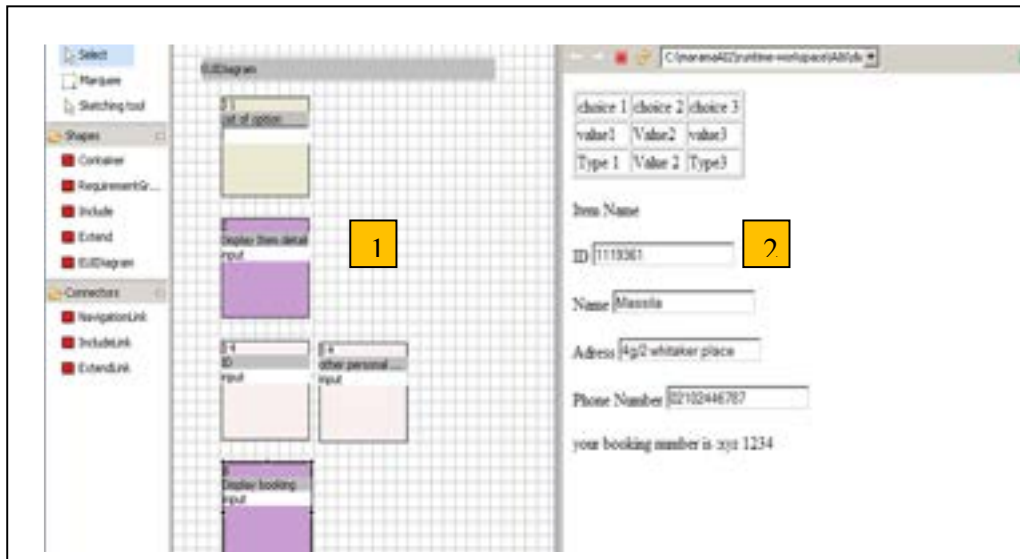


Fig 11. The generated EUI prototype (1) and translated HTML form (2)

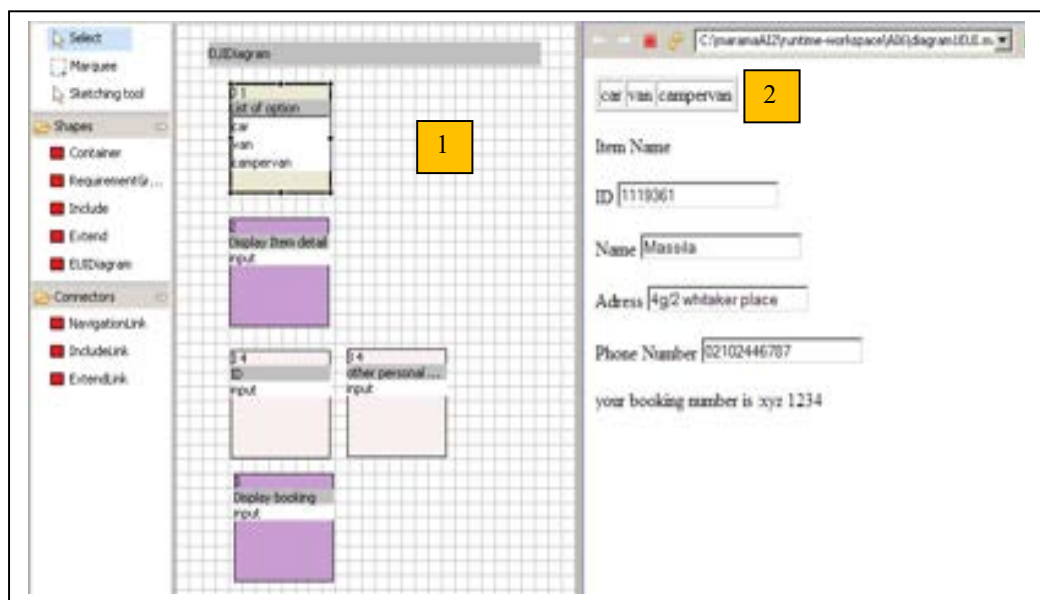


Fig 12. Modifications in Prototypes

From the walkthrough, John thinks that the EUI component of “List of options” is a bit vague and would be better understood by adding detail of the types of options such as “car, van and campervan” as shown in Figure 12 (1). Nancy modifies that on the spot and then shows the result in a HTML form as in Figure 12 (2). Next, she wants to validate and confirm the consistency of her point of view against John’s point of view. She selects one of the EUI components “List of options” (A) and has the tool trace back to the other requirement components: EUC model, abstract interactions and textual natural language requirements as shown in Figure 13. This triggers the associated EUC component and abstract interactions “choose and offer choice” (B) to change colour to red and the essential interactions “indicates, prompts the customer for the pickup and prompts for the type” (C) of the user scenario to be highlighted. Here, Nancy is able to confirm the consistency of all requirement components with John for the earlier collected requirements.

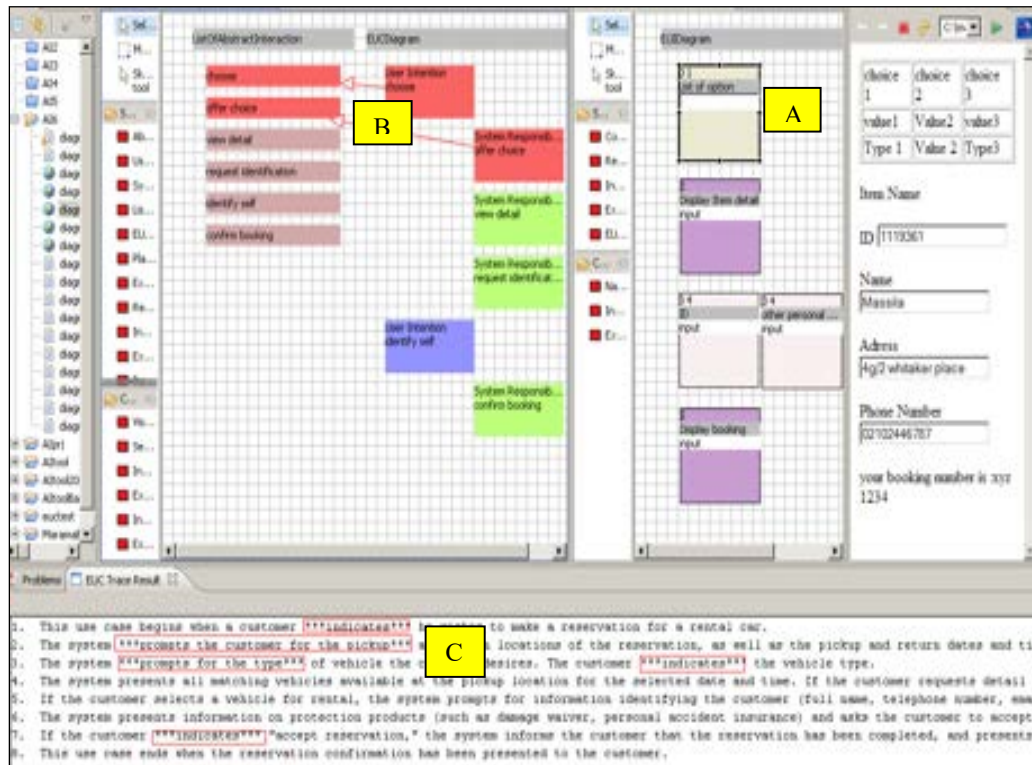


Fig 13. Trace back which performs from the EUI prototype

In summary, Nancy has used the MaramaAIC tool to capture automatically the abstract interactions and to extract the EUCs from the user scenario provided by John. She also used the tool to manage the consistency and to validate the incorrectness and incompleteness of the requirements by using the essential interaction pattern library and “best- practice” template from the EUC interaction pattern library, together with the inconsistency warning, problem marker and highlights. She then sat with John to verify and confirmed further the consistency of the requirements by having the tool generate the prototypes: EUI prototype and HTML form.

Architecture and Implementation

MaramaAIC consists of textual natural language requirement, abstract interaction, Marama Essential (EUC diagram) and MaramaEUI (EUI prototype model) editors. The architecture of Marama AIC is shown in Figure 14. MaramaAIC was realised using the Marama meta-toolset (Grundy et al. 2008) , which is built using the Java–Eclipse platform [Steps 1-2 in Figure 12]. MaramaAIC editors are specified using Marama shape, meta-model and view tools. Each editor is then implemented by interpreting the specification using a set of Marama plug-ins [Step 2]

The meta-model and Domain Specific Visual Language (DSVL) specifications were also supplemented with event handlers to provide low-level model constraints, consistency management support, mapping and interfaces to other elements of the architecture as well as to generate the prototype model (3-7). These were implemented in Java and include generation of dialogues and problem markers to help the user to track, tolerate and resolve the inconsistencies. The event handlers are the vital agent in maintaining consistency among the four forms of requirements components: textual natural language requirements, abstract interaction, EUC diagram and EUI prototype model. An Eclipse text

editor is used to capture natural language requirements and “event handlers” [Step 3] called Trace were implemented to realise extraction of abstract interactions and EUC models from the natural language text. This EUC extractor generates an editable Marama EUC diagram. An MS Access database of mappings of essential interactions to abstract interactions is used in this extraction process. Source natural language phrase to EUC element mappings are recorded with the EUC elements during the extraction process. This allows tracing between these elements when the MaramaAIC user clicks on an item in each view. The “trace back” event handler [Step 3 and 7] uses these mappings to visually highlight the linked natural language phrases, EUC elements and EUI prototype respectively.

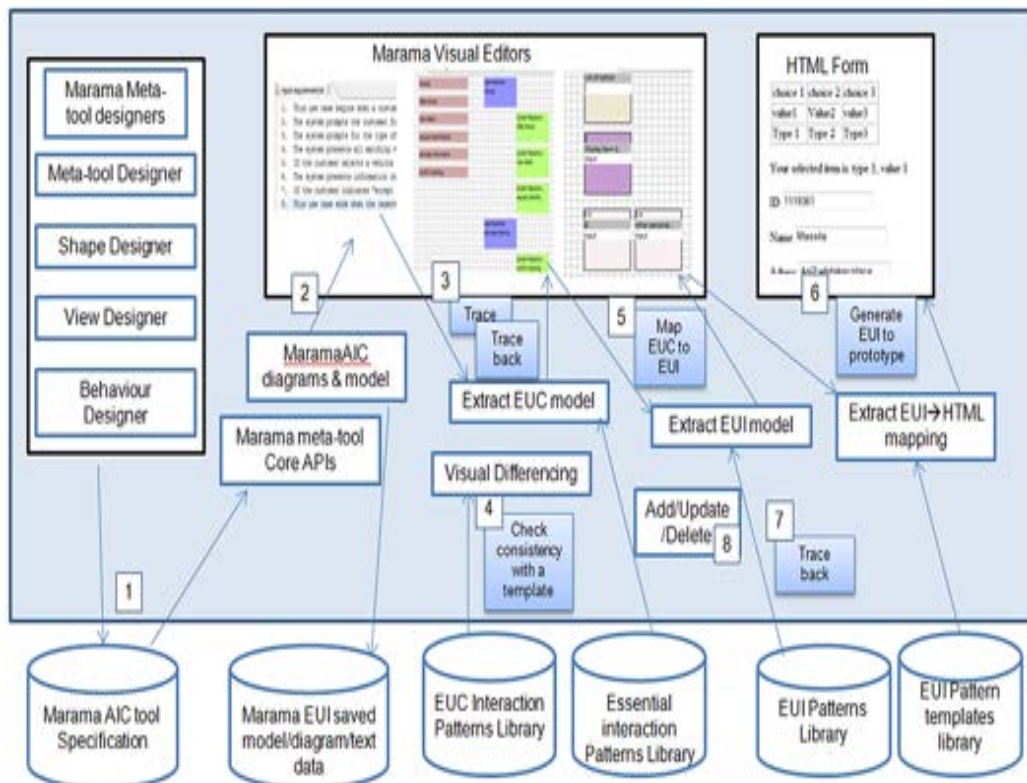


Fig 14. Architecture of MaramaAIC.

A “visual differ” [Step 4] is used to highlight the differences between “best-practice” pattern template and EUC. This often highlights incomplete and/or incorrect sequences, elements, missing elements, or mistyped elements in the extracted EUC, helping the MaramaAIC user to identify problematic requirements.

Another event handler generates an EUI model from the EUC model [Step 5]. This uses a EUI pattern library to map EUC elements to best-fit EUI elements. This EUI model can then be used to generate an HTML form representing a rapid prototype of a form-based interface to the requirements [Step 6]. Updates to any of the models (natural language, abstract interactions, EUC elements or EUI elements) are detected as they are made [Step 8]. These changes are propagated to related elements in the other models. Some changes can be automatically applied. Others are ambiguous so the tool informs the user of the change(s) so the user can make appropriate manual updates. To illustrate further how the event

handlers work in our tool, sequence diagrams are used to demonstrate the interaction. Figure 15 and Figure 16 show an example of interaction of TraceBack and IndexChecker event handlers in operation.

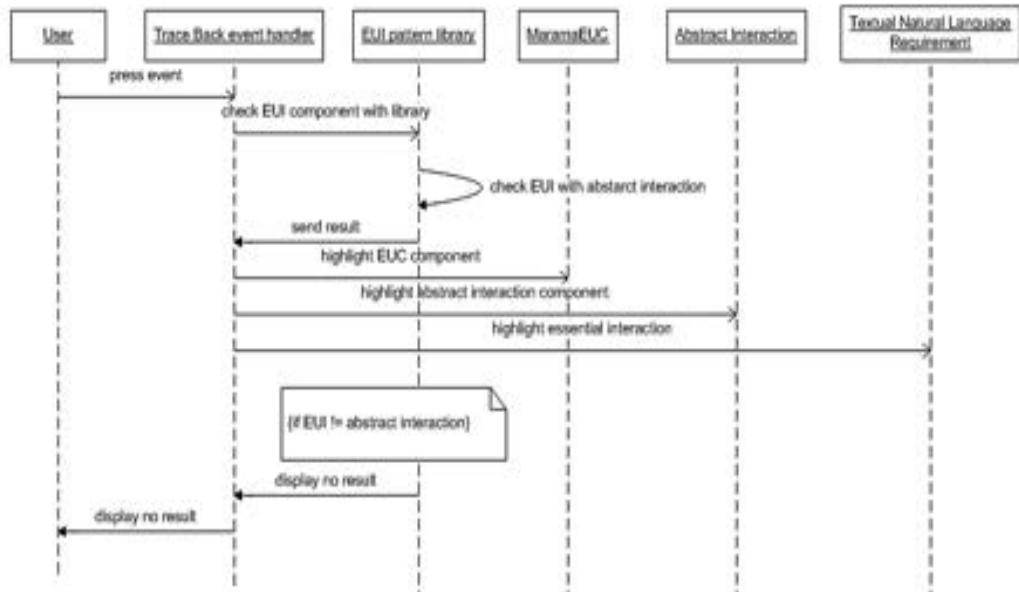


Fig 15. Example of TraceBack interaction from EUI prototype to EUC Model

Figure 15 shows how the user traces back from the EUI prototype component to its source using the TraceBack function. The selected EUI prototype component is analysed by the tracing engine and then matched with the abstract interaction in the EUI Pattern library. If we try to trace back the EUI component, the tool will show where the associated abstract interaction, EUC model and essential interaction for that particular EUI prototype come from. If a newly added component of the EUI prototype does not match an abstract interaction in the EUI Pattern library, no result is provided.

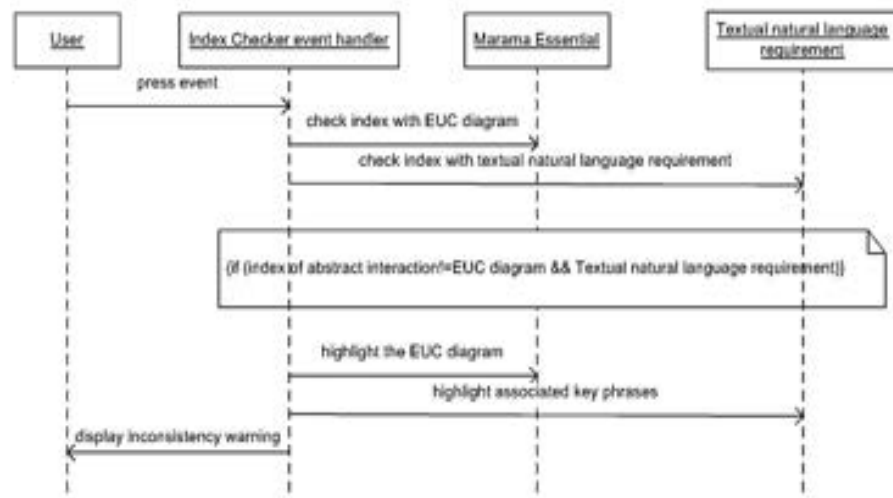


Fig16. Example of Index Checker interaction of Abstract Interaction

Figure 16 shows the function IndexChecker which acts as a checker for the consistency of the sequences in both abstract interaction and EUC Diagrams in Marama AIC. The Index Checker checks the index and location for each abstract

interaction and EUC component. Both need to be in sequence with ordering consistent with the textual natural language requirements. If there is any change of the sequence or location for either, the event handler highlights the associated components either the EUC component or the essential interactions and provides a warning about the inconsistency that has occurred.

Evaluation

Recall that our aim of this work was to determine whether automated tool support for EUC-based requirements capture and validation would improve on manual methods, captured by our research question of “*can automated support for Essential Use Case and Essential User Interface modelling enhance the consistency management and validation of requirements over manual methods?*”.

We conducted three studies in order to evaluate our tool’s efficacy, performance and usability. The first study was on the efficacy and performance of our tool to extract abstract interaction for EUCs. Results were then compared with the preliminary study on the manual extraction process.

The second study was of our tool’s usability and user perceived strengths and weaknesses. Here, participants explored the tool facilities for capturing and checking the consistency of requirements as well as generating the prototype model. The second study was conducted individually to allow us to observe participants and receive feedback one-on-one from them. Participants were given an explanation and demonstration of how to use the prototype tool and the tasks they needed to perform. A task list and a questionnaire sheet were given to participants before they started using the prototype tool.

The third, qualitative, study evaluated the effectiveness of our end-to-end rapid prototyping approach in improving the dialogue between REs and their clients and in improving requirements’ quality. Here, we interviewed and observed three pairs of industry practitioners, one pair member is an industry-based software practitioner experienced in handling software requirements and the other an industry based practitioner experienced in the role of being a client or stakeholder in a software project. This study aimed to understand whether the tool was effective in supporting round trip requirements engineering and validation between REs and their clients.

Efficacy evaluation

We first compared the accuracy of MaramaAIC against the previous results of preliminary study on the manual extraction of essential requirements by 11 novice requirements engineers, as shown in Table 5. MaramaAIC succeeded in identifying almost all the abstract interactions, failing to detect one abstract interaction, providing an accuracy of almost double the participants’ average and better than all but one of the participants’ accuracy. The correctness ratio for manual extraction is only 47% while MaramaAIC’s is 83%. The single error from the tool is because of its failure to detect one of the abstract interactions (Take Cash).

Table 5 Comparison of Manual Extraction and Automated Support of MaramaAIC

Answers	No. Correct answers	
	Manual extraction	Automated Tracing
Identify user	5	1
Verify Identity	4	1
Offer cash	4	1
Choose	6	1
Dispense cash	9	1

Take cash	3	0
Correctness ratio	47%	83%

In order to determine the scalability and efficacy of our tool, we further evaluated its accuracy by applying it to extract EUC models for 15 use case scenarios derived from different researchers, developers and ourselves across a variety of different domains: Online CD catalogue, Cellular phone (Constantine 1998), Voter registration (Stephane 2005) Cash withdrawal (Bjork 2005) Online book (Glinz 2000), Checkout book (library) (Denger, Berry & Kamsties 2003), Seminar Enrolment (Nuseibeh, Easterbrook & Russo 2000), Transfer transaction (Bjork 2005), Deposit transaction ((Bjork 2005), Assign report problem (Horton 2009), Create problem report (Horton 2009) , Report problem (Horton 2009), Booking room (Kim 2006) and Place order (Scenario examples, 2009). The tool correctness was evaluated by comparing the answers with oracle EUC models provided by Constantine and Lockwood (1999), Biddle et al. (2002) and also with models we developed following Constantine and Lockwood's methodology. Correctness ratios for the abstract interactions identified, calculated as they were for the manual extraction study, are shown in Table 6.

Table 6 Efficacy Evaluation on the Extraction Process using MaramaAIC

No.	Requirement	Numbers of Abstract Interaction	Manual results: List of abstract interaction	Automated results: List of abstract interaction	Numbers traced	Ratio
1	Online cd catalog	5	1.view list	✓	5	5:5
			2.search item	✓		
			3.view details	✓		
			4.make order	✓		
			5.calculate cost	✓		
2.	Cellular phone	3	1.make call	✓	2	2:3
			2.receive call	x		
			3.answer call	✓		
3.	Cash withdrawal	6	1.choose account type	✓	4	4:6
			2.select amount	✓		
			3.verify amount	x		
			4.view problem	✓		
			5.verify transaction	✓		
			6.notify result	x		
4.	Online book	7	1.select item	✓	6	6:7
			2.make payment	✓		
			3.ask help	✓		
			4.notify confirmation	x		
			5.verify user	✓		
			6.print invoice	✓		
			7.sent item	✓		
5.	Voter registration	6	1.select option	✓	6	6:6
			2.request identification	✓		
			3.identify self	✓		
			4.check status	✓		
			5.provide identification	✓		
			6.display error	✓		
6.	Borrow book	7	1.verify user	✓	3	3:7
			2.display option	x		
			3.select option	x		
			4.check item	✓		
			5.identify item	x		
			6.print slip	✓		
			7.display message	x		
7.	Checkout book(library)	6	1.identify user	x	5	5:6
			2.verify user	✓		
			3.validate item	✓		
			4.print receipt	✓		
			5.receive receipt	✓		
			6.return item	✓		
8.	Enrollment seminar	9	1.identify self	✓	8	8:9
			2.verify user	x		

			3.display option	✓		
			4.make selection	✓		
			5.check the schedule	✓		
			6.calculate cost	✓		
			7.enroll	✓		
			8.ask payment	✓		
			9.print bill	✓		
9.	Transfer transaction	6	1.select option	✓	6	6:6
			2.chooses account type	✓		
			3.select amount	✓		
			4.provide identification	✓		
			5.verify user	✓		
			6.print receipt	✓		
10.	Deposit transaction	6	1.select option	✓	6	6:6
			2.chooses account type	✓		
			3.select amount	✓		
			4.provide identification	✓		
			5.verify user	✓		
			6.print receipt	✓		
11.	Assign report problem	4	1.select option	x	2	2:4
			2.display result	✓		
			3.select member	x		
			4.confirm status	✓		
12.	Create problem report	7	1.select option	x	4	4:7
			2.request report	x		
			3.create report	x		
			4.save identification	✓		
			5.confirm status	✓		
			6.insert description	✓		
			7. save report	✓		
13.	Report problem	6	1.identify self	x	5	5:6
			2.display help	✓		
			3.select help option	✓		
			4.request description	✓		
			5.describe problem	✓		
			6.offer possible solution	✓		
14.	booking room	4	1.select option	x	3	3:4
			2.select item	✓		
			3.identify self	✓		
			4.print slip	✓		
15.	Place order	6	1.identify self	x	4	4:6
			2.select product	✓		
			3.provide detail	✓		
			4.make payment	✓		
			5.verify information	✓		
			6.confirm order	x		

This shows some variability across the range of scenarios, averaging approximately 80% correctness for extracting abstract interactions. The automated tracing tool does not (and cannot) produce 100% correct answers due to the inherent incorrectness and incompleteness of textual requirements. This is due to various linguistic issues, such as phrases or sentences using a passive pattern, existence of parentheses and grammar issues such as incorrect use of plural or singular, adjectives or adverbs (Tjong, Hallam & Hartley 2006). These problems, however, also lead REs to misunderstand requirements and can be one of the reasons why different requirements engineers or users provide inconsistent results. An average 80% extraction accuracy is lower than desirable, however two points need to be made. Firstly, the accuracy is much better than for manual extraction. Secondly, many of the inaccuracies are picked up when the extracted EUC models are matched against best practice EUC patterns in downstream use of the toolset. This, in turn, can help, via use of the MaramaAIC traceability tooling support, to identify grammatical problems with the textual requirements that cause inaccurate extraction of EUC elements.

Usability study

In our preliminary study, we demonstrated that end users find manual derivation of EUC and EUI prototypes to be difficult, time consuming and error prone. We wanted to demonstrate the effectiveness of our new automated tool support using EUC modelling and EUI prototyping together to support end-to-end rapid prototyping consistency management and validation of requirements. To this end we conducted a user study to evaluate perceptions of the tool and its application.

Participants in this study were 20 software engineering post-graduate students. Their experience as requirements engineers can be categorised as novice to intermediate. Each participant was given a brief tutorial on how to use the tool and some examples of how the tool captures requirements using EUC modelling and EUI prototyping. They first captured the requirements using EUC models and then derived an EUI prototype from the EUC model and natural language requirements. They then mapped the EUI prototype to a concrete HTML-based UI view. Further exercises modifying the EUI prototype followed: adding and deleting EUI components and exploring the result of the modifications in the concrete UI view. We observed the participants' performance while using the tool to accomplish the provided tasks. Participants were asked to think aloud and provide suggestions to enhance the tool. Once all tasks were completed for each part, they were required to answer a questionnaire. Participants completed the questionnaire at their own pace without supervision. The response data were then collected for analysis. Each participant took less than one hour to perform the evaluation. The questionnaire comprised two parts, examining 1) usability and 2) a Cognitive Dimensions (CD) (Blackwell 2001) based assessments. Each question was recorded using a five part Likert scale: 1=strongly disagree to 5=strongly agree.

For Usability criteria, we used the set of criteria suggested by Lund (1998) in the USE questionnaires. The author suggested four criteria that are correlated to one another - Usefulness, Ease of Use, Ease of Learning and Satisfaction (Lund 1998). We used these criteria in developing our questionnaires. We define the criteria as follows.

- Usefulness: how useful the tool is to help users be effective in accomplishing the given task
- Ease of Use: how easily users can work with the tool's user interface and functionality
- Ease of Learning: how easily the user can understand and learn to use the tool
- Satisfaction: is the user satisfied with the tool's capability in performing the required tasks.

The questionnaire comprised several questions for each criterion, which were averaged and converted to a percentage.

We used the Cognitive Dimensions (CD) framework operationalised by Blackwell (2001) in our questionnaires to allow us to explore in detail the reason for each of the user's perceptions for our MaramaAIC tool. CD (Blackwell et al. 2001) is applied here, as it is a common approach for evaluating visual language environments. It helps non-HCI specialist and ordinary users to evaluate usability (Blackwell 1998). In addition, it is lightweight and allows reasoning about usability tradeoffs (Blackwell 1998). In our questionnaire each CD dimension was evaluated by one question. The questions used are adapted from (Kutar 2000). In total, there were ten questions as shown in Table 7.

Table 7 CD Notations Used and Questions Evaluating Them

Cognitive Dimension	Question
Visibility	It is easy to see various parts of the tool

Viscosity	It is easy to make changes
Diffuseness	The notation is succinct and not long-winded
Hard mental effort	Some things do require hard mental effort
Error-proneness	It is easy to make errors or mistakes
Closeness of mapping	The notation is closely related to the result
Consistency	It is easy to tell what each part is for when reading the notation
Hidden dependencies	The dependencies are visible
Progressive evaluation	It is easy to stop and check my work so far
Premature commitment	I can work in any order I like when working with the notation

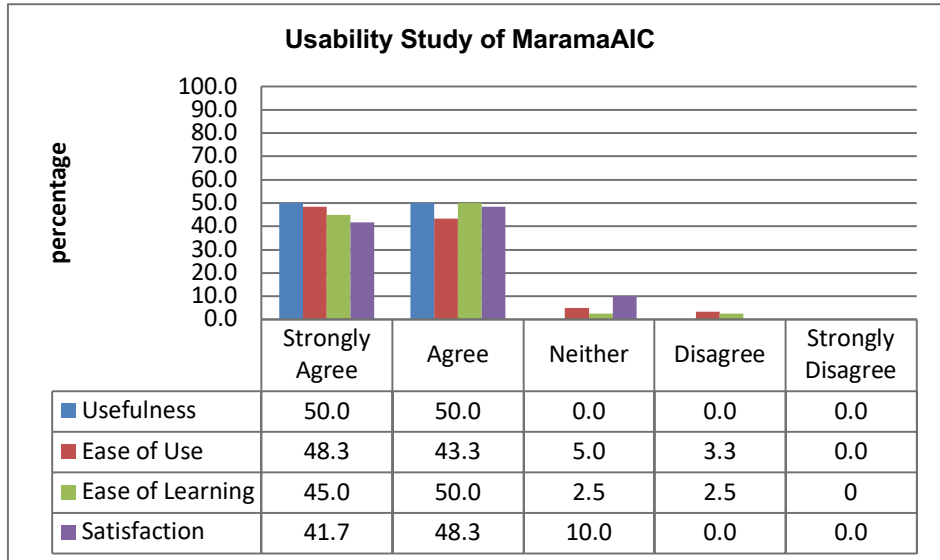


Fig 5. Usability results on MaramaAIC

Figure 17 shows the results of the usability survey conducted for MaramaAIC. For each characteristic, the results of each corresponding question block were averaged to produce the results shown. The results are overall positive with strong agreement from the users over the usefulness of the tool (100% strongly agree or agree on its usefulness), the ease of use (over 90%), ease of learning (95%) and satisfaction (90%). The small number of cases of disagreement over ease of use and ease of learning related to a preference by those participants to have a more descriptive label for each colour and shape used in MaramaAIC. However, with the small number of experimental subjects the results should be viewed as encouraging but not definitively answering our research question.

The CD study allows us to explore in more detail the reasons for these user perceptions. We used the dimensions and questions in Table 7 for this study. The results are based on percentages, reflecting the number of participants' answers for each scale. Figure 18 shows the evaluation results for each of these questions. We believe these results demonstrate interesting usability dependencies between the dimensions that we feel have contributed to the strong usability acceptance of our MaramaAIC.

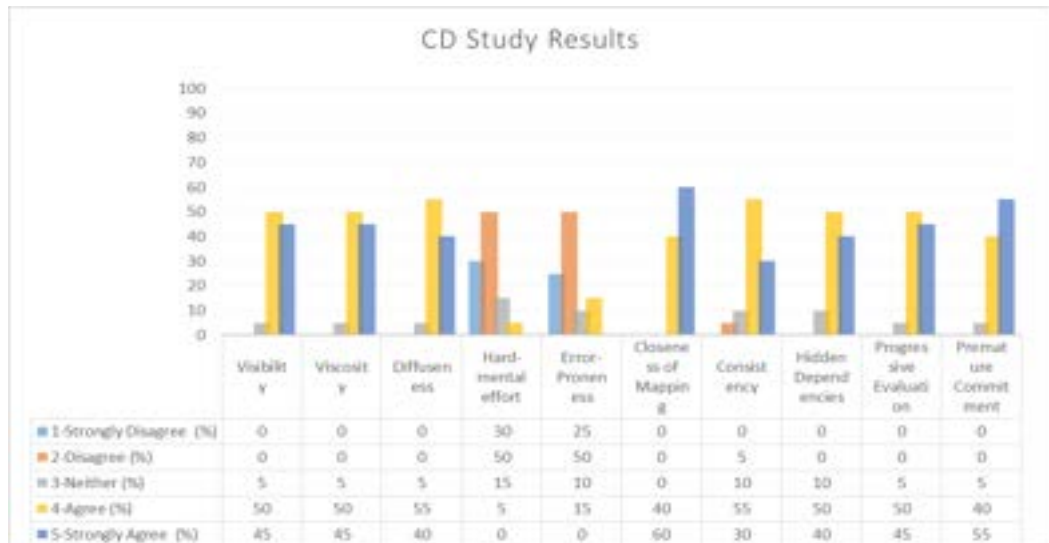


Figure 18 CD Study Results

A summary of the results for each dimension are as follows:

Visibility: Visibility was rated highly, due to explicit use of juxtaposition and the visible trace links. Our speculation is that this is because this “completes the picture” for users.

Viscosity: Participants found it is easy to make changes to the diagrams either in the EUC model or EUI prototype model.

Diffuseness: The notations used by MaramaAIC are succinct and understandable by end users.

Hard-mental effort: MaramaAIC does not need a lot of mental effort to solve the tasks. The tool is able to automatically detect inconsistencies in the requirements and automatically generate the various models.

Error-Proneness: Users disagree that the tool leads the user to errors. This is because all the errors are detected automatically and they could automatically generate the prototype. The EUC and EUI prototype generated is based on the pre-defined pattern library. Thus, this assures the accuracy of the EUC and the UI

Closeness of Mapping: The notations used by the tool are relatively intuitive and understandable. However, the Marama layout mechanism sometimes confused users as to which notation was being used when doing visual diffing.

Consistency: Some of the users were confused when differentiating the notation used to represent the differences between the generated EUC model and the EUC template models.

Hidden dependencies: This rated highly as dependencies among the three requirements components and prototype are made visible using highlighting.

Progressive Evaluation: This rated highly. MaramaAIC allows end users to easily stop and check their work at any time and to changes to be made to any of the requirement components. Thus, end users do not have to worry about the errors as the tool provides an automated support if any errors such as inconsistencies, incompleteness and incorrectness exist.

Premature Commitment: This dimension, which also rated highly (i.e. users regarded the system as having low premature commitment), reflects the sequence of using the tool in order to achieve the results. The tool allows a user to perform the task from any direction. End users can capture requirements or

make changes in any of the components either from the MaramaEUI editor or MaramaEssential editor with consistency maintained.

To summarise, the usability dependencies between the dimensions show that high closeness of mapping and visibility as well as low viscosity assists with issues of hard mental operations and hidden dependencies but somewhat surprisingly did not reduce participants' impressions of error proneness. The high progressive evaluation and low premature commitment contribute to low viscosity.

Use of MaramaAIC by Requirements Engineering Professionals

In our third evaluation, we conducted a qualitative study using pairs of participants, one an industry-based software practitioner experienced in handling software requirements and the other an industry based practitioner experienced in the role of being a client or stakeholder in a software project. Three pairs of participants were recruited. Table 8 shows the background of the participants involved. Pairs of participants were given an explanation and demonstration of our MaramaAIC tool and some requirements extraction, tracing, consistency checking and UI prototyping tasks to be performed.

Table 6 The Participants' Background

Evaluation	Participants (RE=Requirements Engineer, C=Client)	Position	Level of Experience in Requirements	Years of experience in Requirements	Client Background
1	RE 1 & C1	Software Engineer	Intermediate	5 years	Government client with IT Background
2	RE 2 & C2	Staff engineer	Advanced	5 years	Private client with IT Background
3	RE 3 & C3	Senior System Analyst	Advanced	4 years +	Private client with IT Background

Each participant needed to capture textual requirements from the client, map these to an EUC model, and then map them to a UI rapid prototype model. They then showed the results of this to the client participant. Any changes or modifications requested by the client were carried out by the RE using MaramaAIC. We observed the participants carrying out these tasks and video-recorded them to enable us to more closely analyse how the tasks were performed. The participants were also asked to think aloud and express their opinions about the tool. At the end of the session they were asked to answer questions in a semi-structured interview covering the topics of whether the approach helped to improve the dialogue between the RE and client and whether it helped to improve requirements quality.

From our observations and interviews, we found that MaramaAIC assisted both REs and clients to discuss, to confirm and to validate the target system requirements. In evaluation 1, RE1 stated that the tool encouraged her to ask the client to confirm and validate the consistency and correctness of the requirements that she had captured in EUC model. An extract from the dialogue is as follows:

RE1: "So, here is the picture of your requirements. What do you think?"

C1: "All looks good but this component ("list of option") is not necessary."

RE1: "Ok. Let's delete the "list of option" and let us see the prototype."

RE1 deleted the component as requested and then showed C1 the textual requirements and EUC, EUI prototype model and HTML form generated before.

C1 is then requested to validate and confirm the modified requirements against the original requirements and responded:

C1: "yes. I think it is fine now."

A similar dialogue occurred in evaluation 2:

RE2: "This is the outcome of your requirements. Can you please have a look on the prototype to confirm that I'm on the right direction."

C2: "I think something is not right here. I think I need to add a component ("list") to the prototype. Can you please show me the original requirements that I gave before?"

RE2 showed C2 the original requirement in the textual editor written in NL and then made changes by adding the component "list" as requested and then asked C2 to validate the modified requirements.

RE2: "Here are the original requirements and this is the result of the new one. I think this component ("list") is not right to be here. Do you still want it to be added?"

C2: "I think you are right. Delete the list and keep the requirements as it is."

In the case of evaluation 3, RE 3 stated that the tool helped her to visualise the interaction and the outcomes of her captured requirements via the EUC model and prototype model with the client (C3). A dialogue similar to the previous two eventuated:

RE3: "Cool! The tool shows me the interactions between user and system and the prototype. So, sir, here is the picture of your requirements. What do you think? "

C3: "Cool. But I think I need to add a button ("delete") here".

C3 asked the RE to add a component (delete) at the end of the page. RE3 made the changes as requested.

RE3: "Ok. Let's see if it fits with your original requirements (while tracing it back to the textual requirements). It seems fits well here. I think I agree with you"

C3:"Thank you. Everything is perfect now."

In all three cases the tool helped both clients and REs to check the consistency, correctness and completeness of the requirements against the client's original intentions allowing them in real time to explore, discuss and agree or disagree with changes made to the requirements. MaramaAIC helped to both ease and speed up the process of requirements validation through its fast feedback on the impact of changes or modifications.

Overall, the evaluation and interviews with the participants provided positive results. All the REs stated that that the tool helped them to communicate and discuss uncertainty and problems with the clients as well as to confirm and show the results of the requirements to the clients. They were also happy with the explanation and arguments from the clients as they could visualise the results using the prototype. They commented that they did not need to wait for a long cycle of meetings with clients to confirm requirements. Client participants all agreed the tool helped them to clearly identify any errors and misunderstandings and communicate them to the RE. They liked the fact that changes were able to be made immediately and their effects visualised at the same time. This gave them confidence that their requirements were correct, complete and consistent.

In summary this study found that MaramaAIC was able to enhance the quality of dialogue between a RE and client by showing the results of the captured and analysed requirements. The fast feedback and early validation by both parties contributed to better quality of the captured requirements.

Discussion

Our original research question was “*can automated support for Essential Use Case and Essential User Interface modelling enhance the consistency management and validation of requirements over manual methods?*”. We have answered this research question by developing MaramaAIC an automated toolset for EUC and EUI modelling and evaluating it via three quite different studies: we examined the tool’s efficacy and performance in comparison to manual modelling approaches; the tool’s usability and user perceived strengths and weaknesses for end-to-end rapid prototyping support; and finally the effectiveness of the tool in improving the dialogue between REs and clients to improve captured requirements quality.

Our studies showed positive results especially in terms of tool usefulness. They show a good degree of acceptance by end-users of the tool in automatically managing the consistency and validating requirements. Our results also appear to complement prior studies in applying EUCs (Kamalrudin, Grundy & Hosking 2010), (Biddle et al., 2000). It was found by our subjects that our MaramaAIC provides better accuracy and takes lesser time than the manual extraction of EUC from the textual natural language requirements. It is able to detect many quality errors when the extracted EUC models are matched against the best practice EUC patterns. In this case, the detected errors are notified to the users using inconsistency warnings, problem markers and highlights. It was also demonstrated that our tool is able to assist both the RE and clients in the discussion, confirmation and validation of the captured requirements. Our tool is also able to ease and fasten the process of requirements validation via the end-to-end rapid prototyping and the visualisation of effects that help to trigger fast feedback based on any impact of changes or modifications. As noted earlier, however, while encouraging these results can not be viewed as definitively answering our research question due to the limited number of test subjects (20 students and 6 professionals) and limited size and number of exemplar requirements used in the experiments.

However, there are some limitations on the functionality of the tool that requires enhancement. First, we found some problems when dealing with multiple requirements. Although the tool is able to support multiple requirement as described in the section tool usage example, the tool cannot perform a simultaneous traceback for both requirements. Secondly, it is able to perform trace back for one set of requirements at a time only. This somehow makes it difficult for the users to traceback the association of EUCs and EUIs model with the textual natural language requirements. Finally, the tool does not support partial selection of the change, although it provides highlights and an inconsistency warning for inconsistency detection that appear together with the options to either delete or cancel. Thus, this somehow affects the decision of validating the requirements.

Therefore, there are some improvements needed to improve the usability of MaramaAIC. We need to enhance layout to reduce the consistency issues noted and provide training material. The colour and shapes used in the tool need some improvement with better labelling to explain the features. The tool could also be integrated with a GUI template for the generated HTML form for each domain of application. Then, we need to improve the traceability support for multiple requirements where the tool should allow traceback for multiple requirements at the

same time. Further, we also need to consider to enhance the tool by supporting partial selection on changes during the process of validation. The library for essential interaction patterns, EUC interaction patterns and EUI patterns also need expansion. To assist this, a pattern template editor needs to be developed to allow rapid authoring and update of the patterns to be done by any RE.

We believe our preliminary evaluation has shown that our end-to-end approach is a promising way of improving the dialogue between the REs and their clients. However, we need to conduct a longitudinal study to confirm this in extended practice. Other improvements include incorporating better NL processing support to complement our current abstract interaction extraction approach. In addition, we are exploring multi-lingual requirements capture and consistency management via EUCs built on our current end-to-end rapid prototyping approach (Kamalrudin, Grundy & Hosking 2012)

Summary

Inconsistency, Incorrectness and Incompleteness are common errors that always occur in requirements. Besides, there is also limited tool support that able to provide end-to-end support in validating and managing the consistency of requirements between the requirements engineers and their client. We have described an automated tool support called MaramaAIC using semi-formal models: Essential Use Cases (EUCs) and Essential User Interface (EUI) for managing requirements consistency and validation. This tool can automatically extract abstract interactions and EUC models from textual natural language requirements. Then, an EUI prototype model and concrete UI prototype can also be automatically generated from the EUC model. We have also demonstrated that these automation processes perform better than manual processes conducted by requirements engineers. In addition, our tool helps to automatically capture the essential requirements, check for the inconsistency, incorrectness and incompleteness using the developed essential interaction patterns and EUC interaction patterns with the traceability and visualisation support. Our tool is also able to automatically generate UI prototypes using the developed EUI patterns library, which helps to provide a clearer picture of the requirements to the client and help to ease the process to confirm the consistency of the requirements captured by the requirements engineers against the client's original requirements.

Acknowledgement

We acknowledge the support of the participants in our evaluation studies who willingly gave their time. Massila Kamalrudin acknowledges financial support from the University of Auckland, Swinburne University of Technology, Ministry of Higher Education Malaysia (FRGS/F00185) and Universiti Teknikal Malaysia Melaka (UTeM) for their assistance in this research. All authors acknowledge the support of the New Zealand Ministry of Business, Innovation & Employment via funding for the Software Process and Product Improvement project. We also thank Jun Huh for his assistance in developing MaramaAIC and Mark Young for his kindness in providing us the exemplar requirements. Finally, we thank the extremely thorough and detailed comments of the anonymous referees who went above and beyond the call of duty to give us very precise, detailed and very helpful assistance on earlier drafts of this article.

References

- Am, Sampaio, R., Chitchyan, R., Rashid, A. & Rayson, P.: EA-Miner: a tool for automating aspect-oriented requirements identification. Paper presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA (2005)
- Ambler, S. W.: Essential (Low Fidelity) User Interface Prototypes. <http://www.agilemodeling.com/artifacts/essentialUI.htm> (2003-2009). Accessed 20 April 2010
- Ambler, S. W.: The Object Primer: Agile Model-Driven Development with UML 2.0 (3rd ed.), New York Cambridge University Press (2004)

- Bjork, R. C.: Use Cases for Example ATM System. http://www.math-cs.gordon.edu/courses/cs320/ATM_Example/UseCases.html (June 1998). Accessed February 2009
- Biddle, R., Noble, J. & Tempero, E.: Essential use cases and responsibility in object-oriented development. *Aust. Comput. Sci. Commun.*, 24(1), pp. 7-16, (2002)
- Biddle, R., Noble, J. & Tempero, E.: Pattern for Essential Use Cases (C. science, Trans.) (Vol. CS-TR-01/02). Wellington, New Zealand: Victoria University of Wellington (April 2000)
- Blackwell, A., Britton, C., Cox, A., Green, T., Gurr, C., Kadoda, G. & Young, R.: Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In M. Beynon, C. Nehaniv & K. Dautenhahn (Eds.), *Cognitive Technology: Instruments of Mind*, vol. 2117, pp. 325-341. Springer Berlin / Heidelberg (2001)
- Blackwell, T. G. a. A.: Cognitive Dimensions of Information Artefacts: a tutorial. Version 1.2. <https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf> (1998)
- Buskirk, V. R. & Moroney, B. W.: Extending prototyping. *IBM Systems Journal*, 42(4), pp. 613-623., (2003)
- Constantine, L. L.: Rapid Abstract Prototyping Software development 6 (11), 1998
- Constantine, L. L., & Lockwood, L. A. D. : Software for use: a practical guide to the models and methods of usage-centered design: ACM Press/Addison-Wesley Publishing Co., (1999)
- Corporation, B. S.: CaliberRM™ Enterprise Software Requirements Management System. <http://www.borland.com/us/products/caliber/index.html> (2011). Accessed 08 February 2011
- Cristian, B.: Generating an Abstract User Interface from a Discourse Model Inspired by Human Communication, (2008)
- Dardenne, A., Van Lamsweerde, A., & Fickas, S. (1993). Goal-directed requirements acquisition. *Science of computer programming*, 20(1), 3-50.
- Denger, C., Berry, D. M. & Kamsties, E.: Higher Quality Requirements Specifications through Natural Language Patterns. Paper presented at the Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering, pp. 80.80-7695-2047-7692: IEEE Computer Society (2003)
- Egyed, A.: Scalable Consistency Checking Between Diagrams-The ViewIntegra Approach. Proceedings of the 16th IEEE international conference on Automated software engineering, pp. 387. IEEE Computer Society, (2001)
- Evans, G.: Getting from use cases to code, Part 1: Use-Case Analysis. <http://www.ibm.com/developerworks/rational/library/5383.html>. Accessed January 2009
- Fabbrini, F., Fusani, M., Gnesi, S. & Lami, G.: The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. Paper presented at the Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard, 2001
- Finkelstein, A., & Emmerich, W.: The future of requirements management tools. *Information Systems in Public Administration and Law*, (2000)
- Geisser, M., Hildenbrand, T. & Riegel, N.: Evaluating the Applicability of Requirements Engineering Tools for Distributed Software Development (D. o. I. S. 1, Trans.) *Working Paper 2/2007* (Working Papers in Information Systems ed.). Germany: University of Mannheim., 2007
- Gervasi, V. & Zowghi, D.: Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol.*, 14(3), pp. 277-330. , 2005
- Glinz, M.: A lightweight approach to consistency of scenarios and class models, Proc.4th International Conference on Requirements Engineering 2000, 2000, pp. 49-58., (2000)
- Grundy, J. C., Hosking, Huh, J. & Li, N.: Marama: an Eclipse meta-toolset for generating multi-view environments. Paper presented at the 2008 IEEE/ACM International Conference on Software Engineering, Leipzig, Germany, May 2008
- Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W. & Schwinger, W.: Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 1-42. doi: 10.1007/s10515-012-0102-y
- Horton, T.: Example Use Cases for PARTS. <http://www.cs.virginia.edu/~horton/cs494/examples/parts/usecases-ex1.html>. Accessed February 2009
- Hull, E., Jackson, K. & Dick, J.: DOORS: A Tool to Manage Requirements Requirements Engineering, pp. 173-189. Springer, London (2005)
- Huzar, Z., Kuzniarz, L., Reggio, G. & Sourrouille, J. L.: Consistency Problems in UML-Based Software Development *UML Modeling Languages and Applications*, pp. 1-12., (2005)
- IBM. Rational RequisitePro A requirements management tool. <http://www-01.ibm.com/software/awdtools/reqpro/>. Accessed 13 February 2011

- Inc., S. S.: Serena. Requirements Management The Proven Way to Accelerate Development. <http://www.serena.com/docs/repository/products/rm/wp900-001-0505.pdf> (2011). Accessed 14 February 2011
- Jacobson, I., Booch, G., Rumbaugh, J., Rumbaugh, J., & Booch, G. (1999). The unified software development process (Vol. 1). Reading: Addison-wesley.
- Kim, J., Park, S. & Sugumaran, V. : Improving use case driven analysis using goal and scenario authoring: A linguistics-based approach, *Data & Knowledge Engineering*, vol. 58, pp. 21-46, (2006)
- Kamalrudin, M., Grundy, J. & Hosking, J.,: Tool Support for Essential Use Cases to Better Capture Software Requirements. Paper presented at the 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20-24 September 2010
- Kamalrudin, M. & Grundy, J.: Generating essential user interface prototypes to validate requirements. Paper presented at the Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, 2011
- Kamalrudin, M., Ahmad, S.S., Sidek, S. & Daud, N.: A Review of Requirements Engineering Tools for Requirements Validation Software Engineering Process, *International Journal of Software Engineering, IJSET*, vol. 1, (2014)
- Kamalrudin, M., Grundy, J. & Hosking, J.: MaramaAI: Tool Support for Capturing and Managing Consistency of Multi-lingual Requirements, *27th Automated software Engineering Conference*, Essen, Germany, (2012)
- Kotonya, G. & Sommerville, I.: Requirement Engineering Process and Techniques. West Sussex, England: John Wiley & Sons Ltd, (1998)
- Kovacevic, S. UML and User Interface Modeling The Unified Modeling Language. «UML»'98: Beyond the Notation, pp. 514-514., (1999)
- Kutar, M., Britton, C. & Wilson, J.: Cognitive Dimensions An Experience Report. Paper presented at the Twelfth Annual Meeting of the Psychology of Programming Interest Group, Memoria, Cozenza Italy, (2000)
- Lund, A.: USE Questionnaire Resource Page. <http://usesurvey.com/IntroductionToUse.html> (2009). Accessed February 2010
- Lampert, L. (2002). Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc..
- Lang, M. & Duggan, J.: A Tool to Support Collaborative Software Requirements Management. *Requirements Engineering*, 6(3), 161-172(2001). doi: 10.1007/s007660170002
- Larry, L. C. & Lucy, A. D. L.: Structure and style in use cases for user interface design Object modeling and user interface design: designing interactive systems. pp. 245-279. Addison-Wesley Longman Publishing Co., Inc., (2001)
- Larry, L. C. & Lucy, A. D. L.: Usage-centered software engineering: an agile approach to integrating users, user interfaces, and usability into software engineering practice. Paper presented at the Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, (2003)
- Lopez-Herrejon, R. E. & Egyed, A.: Towards fixing inconsistencies in models with variability. Paper presented at the Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, Leipzig, Germany, (2012)
- Ltd, S. D.: Creative New Media. [http://www.silicon-dream.com/\(1996-2011\)](http://www.silicon-dream.com/(1996-2011)). Accessed 25 May 2010
- Mommel, T., & Reiterer, H.: Inspector: Interactive UI Specification Tool *Computer-Aided Design of User Interfaces VI*. pp. 163-175., (2009)
- Neill, C. J. & Laplante, P. A.: Requirements engineering: the state of the practice. *Software, IEEE*, 20(6), pp. 40-45., (2003)
- Nentwich, C., Wolfgang, E. & Anthony, F.: Consistency management with repair actions. Paper presented at the Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, (2003)
- Nguyen, T. H., Vo, B. Q., Lumpe, M. & Grundy, J.: *REInDetector: a framework for knowledge-based requirements engineering*. Paper presented at the Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, (2012)
- Nuseibeh, B., Easterbrook, S. & Russo, A.: Leveraging Inconsistency in Software Development. *Computer*, 33(4), pp. 24-29., (2000)
- Perrouin, G., Brottier, E., Baudry, B. & Le Traon, Y.: Composing Models for Detecting Inconsistencies: A Requirements Engineering Perspective *Requirements Engineering: Foundation for Software Quality* , pp. 89-103), (2009)

- Reder, A. & Egyed, A.: Incremental Consistency Checking for Complex Design Rules and Larger Model Changes. In R. France, J. Kazmeier, R. Breu & C. Atkinson (Eds.), *Model Driven Engineering Languages and Systems*. vol. 7590, pp. 202-218. Springer, Berlin Heidelberg (2012)
- Robertson, S. & Robertson, J.: *Mastering the Requirements Process (2nd Edition)*: Addison-Wesley Professional, (2006)
- Sardinha, A., Chitchyan, R., Weston, N., Greenwood, P. & Rashid, A. EA-Analyzer: automating conflict detection in a large set of textual aspect-oriented requirements. *Automated Software Engineering*, pp. 1-25. doi: 10.1007/s10515-012-0106-7
- Satyajit, A., Hrushikesh, M. & George, C.: Domain consistency in requirements specification *Quality Software*, 2005. (QSIC 2005). Fifth International Conference on. pp. 231-238. 1550-6002., (2005)
- Scenario examples. <http://www.opensrs.com/resources/documentation/sync/scenarioexamples.htm>. Accessed February 2009
- Some, S. S.: *Use Cases based Requirements Validation with Scenarios*. Paper presented at the Proceedings 13th IEEE International Conference in Requirements Engineering 2005, (2005)
- Tjong, S. F., Hallam, N. & Hartley, M. : Improving the Quality of Natural Language Requirements Specifications through Natural Language Requirements Patterns. Paper presented at the Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference, (2006)
- Yijun, Y.: From Goals to Aspects: Discovering Aspects from Requirements Goal Models. In Proc. 12th IEEE International Requirements Engineering Conference 2004, (2004), 6-11 Sept. 2004.
- Yu, E. S. (1997). Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997.*, Proceedings of the Third IEEE International Symposium on (pp. 226-235). IEEE.
- Yue, T., Briand, L.C., Labiche, Y.: aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models. *ACM Trans. Softw. Eng. Methodol.* 24(3), 13 (2015)
- Yufei, X., Tao, T., Tianhua, X. & Lin, Z.: Research on requirement management for complex systems. Paper presented at the 2nd International Conference Computer Engineering and Technology (ICCET), 2010, 16-18 April 2010, (2010)
- Zisman, G. S. a. A.: *Handbook of Software Engineering and Knowledge Engineering*. In S. K. Chang (Ed.), (Vol. Volume 1, pp. 329-380): World Publishing co., (2001)
- Zhang, G., Matthias M. Hözl: Weaving semantic aspects in HiLA. *AOSD 2012*, 263-274(2012)
- Zhang, G.: Aspect-Oriented Modeling of Mutual Exclusion in UML State Machines. *ECMFA 2012*, 162-177(2012)

3.3 Adaptable, Model-driven Security Engineering for SaaS Cloud-based Applications

Almorsy, M., Grundy, J.C., Ibrahim, A., Adaptable, Model-driven Security Engineering for SaaS Cloud-based Applications, *Automated Software Engineering*, vol. 21, no. 2, April 2014, Springer, pp. 187–224.

DOI: [10.1007/s10515-013-0133-z](https://doi.org/10.1007/s10515-013-0133-z)

Abstract: Software-as-a-service (SaaS) multi-tenancy in cloud-based applications helps service providers to save cost, improve resource utilization, and reduce service customization and maintenance time. This is achieved by sharing of resources and service instances among multiple “tenants” of the cloud-hosted application. However, supporting multi-tenancy adds more complexity to SaaS applications required capabilities. Security is one of these key requirements that must be addressed when engineering multi-tenant SaaS applications. The sharing of resources among tenants—i.e. multi-tenancy—increases tenants’ concerns about the security of their cloud-hosted assets. Compounding this, existing traditional security engineering approaches do not fit well with the multi-tenancy application model where tenants and their security requirements often emerge after the applications and services were first developed. The resultant applications do not usually support diverse security capabilities based on different tenants’ needs, some of which may change at run-time i.e. after cloud application deployment. We introduce a novel model-driven security engineering approach for multi-tenant, cloud-hosted SaaS applications. Our approach is based on externalizing security from the underlying SaaS application, allowing both application/service and security to evolve at runtime. Multiple security sets can be enforced on the same application instance based on different tenants’ security requirements. We use abstract models to capture service provider and multiple tenants’ security requirements and then generate security integration and configurations at runtime. We use dependency injection and dynamic weaving via Aspect-Oriented Programming (AOP) to integrate security within critical application/service entities at runtime. We explain our approach, architecture and implementation details, discuss a usage example, and present an evaluation of our approach on a set of open source web applications.

My contribution: Contribution: Developed initial ideas for the research, co-supervised the two PhD students, co-authored significant parts of paper

Adaptive, Model-driven Security Engineering for SaaS Cloud-based Applications

Mohamed Almorisy · John Grundy · Amani S. Ibrahim

Received: 9 Jul 2012 / Revised 16 Jan 2013 / Revised 12 Jul 2013 / Accepted: 12 Aug 2013

Abstract Software-as-a-service (SaaS) multi-tenancy in cloud-based applications helps service providers to save cost, improve resource utilization, and reduce service customization and maintenance time. This is achieved by sharing of resources and service instances among multiple "tenants" of the cloud-hosted application. However, supporting multi-tenancy adds more complexity to SaaS application's required capabilities. Security is one of these key requirements that must be addressed when engineering multi-tenant SaaS applications. The sharing of resources among tenants i.e. multi-tenancy increases tenants' concerns about the security of their cloud-hosted assets. Compounding this, existing traditional security engineering approaches do not fit well with the multi-tenancy application model where tenants and their security requirements often emerge after the applications and services were first developed. The resultant applications do not usually support diverse security capabilities based on different tenants' needs, some of which may change at run-time i.e. after cloud application deployment. We introduce a novel model-driven security engineering approach for multi-tenant, cloud-hosted SaaS applications. Our approach is based on externalizing security from the underlying SaaS application, allowing both application and security to evolve at runtime. Multiple security sets can be enforced on the same application instance based on different tenants' security requirements. We use abstract models to capture service provider and multiple tenants' security requirements and then generate security integration and configurations at runtime. We use dependency injection and dynamic weaving via Aspect-Oriented Programming (AOP) to integrate security within critical application entities at runtime. We explain our approach, architecture and implementation details, discuss a usage exam-

Mohamed Almorisy, John Grundy, and Amani S. Ibrahim
Centre for Computing & Engineering Software Systems,
Swinburne University of Technology, Melbourne, Australia
E-mail: malmorsy@swin.edu.au, jgrundy.swin.edu.au, aibrahim.swin.edu.au

ple, and present an evaluation of our approach on a set of open source web applications.

Keywords Software-as-a-Service · Model-driven Engineering · Adaptive-Security · Security Engineering · Tenant-Oriented Security

1 Introduction

Software-as-a-service (SaaS) is one of the key service delivery models introduced by the cloud computing model [3]. SaaS simplifies the software procurement process to renting services instead of buying them and their underlying infrastructure. Thus tenants can save infrastructure cost, software license, system administration, and development staff. Tenants can also switch to different service providers. The SaaS model helps service providers to target the small and medium enterprise markets by offering them a reasonable software adoption cost model.

Multi-tenancy is a new SaaS architecture pattern where service tenants share a single service instance. Multi-tenancy helps SaaS service providers to focus on operating, customizing, maintaining, and upgrading a single instance. On the other hand, multi-tenancy increases service tenants' concerns about the security of their outsourced cloud-hosted assets that are shared with other tenants and who may be either competitors or malicious users. Supporting multi-tenancy also adds more requirements on service providers as they have to develop or reengineer their systems to support multi-tenancy and to ensure tenants' data isolation. In addition, actual SaaS application tenants often become known only after applications have been delivered. Tenants usually have different security requirements that emerge at runtime based on their current business objectives and security risks.

Existing, traditional design-time security engineering approaches, such as KAOS [17], UMLsec [15], secureUML [18], focus on how to identify, capture, and model system security objectives and requirements that need to be enforced in the software under development. These approaches focus on mapping security requirements identified in the early-stage of security requirements engineering on system design entities (components, classes, methods, and interactions). Some of these efforts, such as UMLsec [15] support formal security analysis to verify the satisfaction of the specified security properties. Few of these [24] have toolsets that help, very limited, in generating security code or configurations with the system source code based on using model-driven engineering techniques. Most of these efforts [15], [18] do not address how these security requirements are designed and implemented in these systems. Thus, software developers will typically have to build these security requirements together with the system business function implementations. For example, a specified security property using UMLsec for example on a business function will be achieved by adding appropriate security code with the business

function code. By its very nature, this leads quickly to systems with built-in (hardcoded) security capabilities that are often hard to modify [19]. This approach also complicates the integration with third party security controls as it often requires manual effort by security engineers. Hardcoding such security within software systems limits the flexibility to adapt enforced security in order to meet the new security challenges. Integrating systems with built-in security with customers' operational environment security management systems requires reengineering such systems to inject new security integration code. When multi-tenant systems are considered, this hard-coded approach to security further complicates update of security requirements which often differ by tenant. While some techniques have been experimented to address this e.g. using dynamic Aspect-oriented Programming to inject updated security code [24], this is typically disconnected from the security requirements and design models.

In the area of SaaS applications security engineering, we have determined two key problems: maintaining isolation between different tenants' data; and enforcing different tenants' security requirements. The first problem can be easily addressed at design-time as one of the key security requirements. However, the later problem is hard to incorporate at design-time as software tenants and their security requirements emerge at runtime. Thus, we claim that the solution to this problem requires a different security model as we introduce in this paper.

Component-based (CBSE) and service-oriented (SOA) security engineering approaches [13][25] do support late security engineering (deployment time). However, most of these approaches focus on generating security code, using Aspect-oriented Programming (AOP). These approaches benefit from, but are limited by, the underlying application architecture (CBSE, SOA) to deliver flexible and adaptable security. Some adaptive security engineering approaches have been investigated [14, 33, 10]. However most focus on low-level details or limited to specific security attributes e.g. adaptive access control. These efforts require preparing applications at design time to support runtime adaptation. Thus, these efforts cannot be adapted to deliver multi-tenant SaaS application security engineering. New research efforts in securing multi-tenant SaaS applications have focused on: (i) (re)engineering multi-tenant SaaS applications to extend their security capabilities [8, 7]; (ii) maintaining isolation between different tenants' data at rest, at processing or at transmission [12, 29]; and (iii) developing security controls and architectures that deliver SaaS application security (e.g. access control) taking into account multi-tenancy dimension [40, 38]. Most of those efforts depend on or lead to built-in, or predefined, security architectures for SaaS applications. Thus tackling the loss of control concerns raised by cloud consumers – i.e. capturing and enforcing tenants' security requirements, and integration of SaaS applications with tenants' security infrastructure are not addressed before.

Furthermore, existing industrial security platforms such as the Java Security Model, Spring Security Framework (acegi), and Microsoft WIF, provide a

set of security mechanisms that help developers in securing their applications and reducing number of errors that arise from using custom security functions. Using these frameworks requires system engineers to write integration code exposing platforms supported APIs at every critical application entity. The resultant applications still have built-in security and are tightly coupled with the adopted platform. Integration with third-party security controls requires manual development and configuration changes.

A key gap in the multi-tenant application security area is that lack of adaptable security support as well as the lack of multi-tenant security engineering support i.e. to support capturing and enforcing different tenants' security requirements at runtime without a need to conduct application maintenance. We formulate this gap in the following research questions that we tackle in this paper:

- How can we capture different tenants' security requirements for different application or service entities?
- How can we enforce different tenants' security requirements on any arbitrary application or service entity?
- How can we verify that critical entities correctly enforce specified security needs?
- How can we carry out these tasks at runtime, as tenants emerge after application or service deployment?

In this paper, we introduce a novel approach called MDSE@R (Model-Driven Security Engineering at Runtime) for multi-tenant cloud-based applications. MDSE@R supports capturing, enforcing, and verifying different tenants' and service providers' security requirements at runtime without a need to modify/customize the underlying application implementation i.e. it works with both existing and new SaaS applications. Our approach is based on promoting security engineering efforts to be conducted at runtime instead of design time. This is facilitated by externalizing security from the target application. Thus both application and security can evolve at runtime. On the other hand, we automate the integration of whatever security requirements/controls within any application entity that was marked as critical/secure. The list of critical application entities emerge at runtime based on current risks.

Using MDSE@R, service providers deliver a service description model (SDM) as a part of the application delivery package, details are discussed in Section 4. The SDM is a mega-model (a mega-model [35] is a model that contains a set of models and a set of relations between these models) that contains details of the application features, architecture, classes, etc. Furthermore, they deliver a security specification model (SSM). The SSM is a mega-model that contains details of the security to be enforced on application entities (features, components, etc.). Rather than the mandatory security controls (i.e. security controls that cannot be replaced by tenants'), other security controls should not be built-in the delivered application, they rather implemented and deployed externally and weaved with the secured application entities at runtime.

Thus such controls can be updated, replaced, or disabled at runtime without modifying the target application. If this is not the case (i.e. the application is developed with all security controls built-in), we can use our preprocessing tool to disable such controls and deploy them externally [2]). Different sets of security controls can also be enforced at runtime.

During the provisioning of new tenants, the service provider creates an instance of the application or configures the shared instance to disable or enable certain features for the new tenant. This is reflected in the tenant copy of the application or the services description model. This SDM copy is called tenant service description model (TSDM). Tenants can specify their security details using their copy of the service SSM. This copy is called tenant security specification model (TSSM). Tenants can add new security controls or disable/replace/modify the existing security controls. At runtime, these models may be updated to reflect new security needs. These updates are automatically reflected on the target application using MDSE@R. The service provider can specify certain security controls as mandatory (or develop them within the application). This means that such controls cannot be disabled or modified by the application or service tenants. This is very important in enforcing security isolation controls, for example. Best practice patterns of common controls and configurations can be reused for common security needs by either service providers or service tenants.

Getting tenants involved in managing their asset security helps in reducing the lack-of-trust and mitigating the loss-of-control problems that arise from the adoption of cloud services. We have validated our approach on seven significant open source applications. We have conducted a security features evaluation, performance evaluation and user evaluation of our approach and prototype platform . Below we summarize the key contributions of in this paper:

- Model to capture detailed security requirements for different tenants in cloud application.
- Approach of linking security details to points of interest in target cloud application at differing levels of abstraction.
- Approach to take this system and security model and use to enforce requirements on running application.
- Demonstration of its ability to capture & enforce desired security on several third-party applications.

Section 2 begins with a motivating example for our research and identifies key challenges and requirements that must be satisfied by a multi-tenant security engineering approach. Section 3 reviews key related work. Section 4 provides an overview of our MDSE@R approach. Section 5 describes a usage example of our MDSE@R framework and toolset. Section 6 describes our framework architecture and implementation details. Section 7 presents our evaluation results of MDSE@R. Section 8 discusses implications of our approach, key strengths and weaknesses, and areas for further research.

2 Motivating Scenario

In this section, we introduce a simple and typical cloud scenario where a multi-tenant service has been procured by different tenants. Each of them is worried about the data security and has security requirements and controls that need to be applied. Such scenario cannot be satisfied by the existing security models i.e. supporting different security requirements on the same service instance.

Scenario. Consider "SwinSoft", a well-known software house in developing business applications. Swinsoft has recently developed a new cloud-based SaaS ERP solution called "Galactic. Galactic is designed to support both multi-tenant models including single-tenant, single-instance; and multi-tenant, single-instance. SwinSoft hosts Galactic on a cloud platform delivered by GreenCloud (GC). During the development of Galactic, SwinSoft used external services to speed up the application development. These services include: Currency-Now and build workflow services to get up-to-date currency exchange rates and flexible workflow engine (developed and deployed on GC); and Batch-MPRD to conduct transactions' posting using the map-reduce model that improves and parallelizes the batch posting operations (hosted in BlueCloud - another cloud platform).

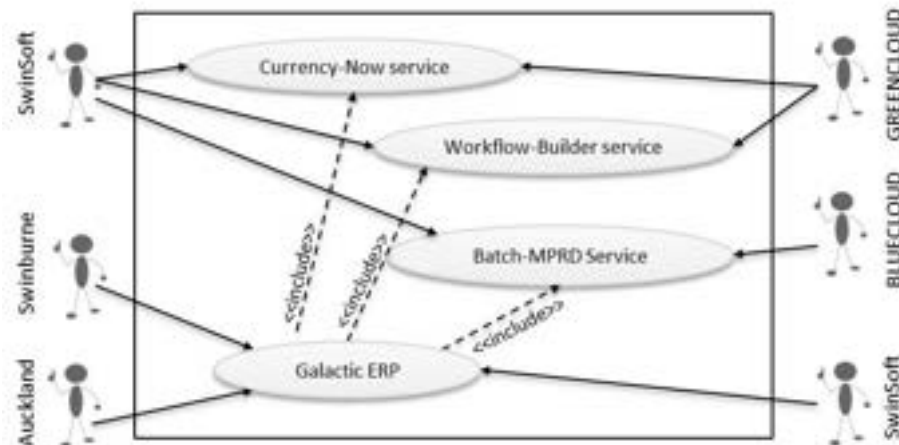


Fig. 1 Use case diagram for the motivating example

Swinburne University is going to purchase a new ERP solution in order to automate its internal process. After investigation of available solutions, Swinburne has decided to go for Galactic ERP solution, to save upfront investment required and keep infrastructure cost optimized. At the same time, "Auck-

land University has also decided to purchase the Galactic ERP application. However, each of these Galactic service tenants have their own, quite different, business functions and security objectives. Swinburne has special security requirements because it is ISO27000 certified. Swinburne security architects conduct periodic risk assessment. This may result in a requirement to reconfigure the enforced applications security to block newly discovered threats. It needs to maintain similar security policies on Galactic as those used in their local environment. This includes using active directory to support Single Sign-On (SSO), applying a role-based access control (RBAC) model on Galactic, their access control policies should consider end-user location and request time, integrity of data transmitted must be maintained, and confidentiality of Swinburne data must be enforced. Auckland assigns high risk to Galactic maintained assets because they are outsourced for hosting on external third-party cloud platform. Thus, they have strong security constraints that are different from their local systems. This includes applying an attribute-based access control (ABAC) model for access control [37], use of a two-factor authentication system, transaction accountability and audit-ability, and all data must be kept confidential. Both organizations thus would like to use the multi-tenant Galactic service while enforcing different security requirements and integrating with different security services.

Key challenges. The analysis of the above scenario identifies the following challenges: security requirements differ from one tenant to another; each tenant's security requirements may change over time based on current operational environment security and business objectives; Galactic security should support integration with each tenant's security controls in order to achieve coherent security solutions; and new security vulnerabilities may be discovered in Galactic application at any time. Using traditional security engineering techniques would require SwinSoft to conduct a lot of application maintenance iterations to deliver application patches that block vulnerabilities and adapt the application to every new customer needs. Multiple versions of the application, one for each tenant and with substantial differing security enforcement embedded in the application, would have to be maintained.

Key requirements. A new security engineering approach that addresses these challenges is needed. It should enable each tenant to specify and enforce their security requirements based on their current security needs. Security should be applied to any arbitrary application entity. No predefined security interception points specified at design time. It should support interception of any applicable application method. Security specification should be supported at different levels of abstraction based on the customers' experience, scale and engineers' capabilities. Integration of security with application entities should be supported at different levels of granularity, from the application as one unit to a specific application method. The security engineering approach should enable integration with third-party security controls. It should support

the application and security specifications to be reconfigured at both design time and runtime.

3 Related Work

Existing academic security engineering efforts have focused on capturing and enforcing security requirements at design time, supporting adaptive security, and multi-tenant security engineering. On the other hand, most industrial efforts have focused on delivering security platforms that can help software developers in implementing their security requirements using readymade standard security algorithms and mechanisms.

3.1 Design Time Security Engineering

Software security engineering aims to develop secure systems that remain dependable in the face of attacks [4]. Security engineering activities include: identifying security objectives that systems should satisfy; identifying security risks that threaten system operation; elicitation of security requirements that should be enforced on the system to achieve the expected security level; developing security architectures and designs that deliver the security requirements and integrates with the operational environment; and developing, deploying and enforcing the developed or purchased security controls. These efforts can be categorized as follows:

- Early-stage security engineering approaches focus only on security requirements elicitation and capturing at design time. KAOS [17] was extended to capture security requirements in terms of obstacles to stakeholders' goals. Obstacles are defined in terms of conditions that when satisfied will prevent certain goals from being achieved. Secure i* [18] focuses on identifying security requirements through analysing relationships between users, attackers, and agents of both parties. Secure Tropos [28] captures details about the security requirements and trust goals, introducing two categories of goals: hard goals that reflect system functional requirements and soft goals reflecting non-functional requirements (security).
- Later-stage security engineering approaches typically focus on security engineering during system design. Misuse cases [34] capture use cases that the system should not allow and may harm the system operation or security. UMLsec [15] extends UML with a profile that provides stereotypes to be used in annotating design elements with security intentions and requirements. UMLsec provides a comprehensive UML profile. However, it was originally developed for use during the design phase. UMLsec has stereotypes for predefined security requirements only (secrecy, secure dependency, critical), , though it is possible to define extensions. Some extensions and applications of UMLsec enable it to be used to support later tasks

of software development e.g. security testing and verification [16?] and runtime verification of history-based properties [5]. A limitation we have found with UMLsec is that it mixes security with system entities at design time, which we have found complicates modifications of system security capabilities especially when they are applied at runtime. SecureUML [19] provides a meta-model to design RBAC policies of target systems. Both approaches are tightly coupled with system design models. Both early and later stage approaches lack a complete security model that captures security details and abstraction levels. Both do not support generating security code that realizes the specified security requirements.

- Security engineering processes include SQUARE [20], SREP [21] and Microsoft SDL. Such processes specify the steps to follow when capturing, modelling, and coding system security requirements. Such processes are aligned with system development processes. Most security approaches and processes focus on engineering security at design time. They often make assumptions about the security of the environments in which an application will operate. It is often difficult to integrate system's security with the operational environment security as software systems depend on their built-in security controls.

3.2 Adaptive Application Security

Several research efforts try to enable systems to adapt their security capabilities at runtime. Extensible Security Infrastructure [14] is a framework that enables systems to support adaptive authorization enforcement through updating in memory authorization policy objects with new low level C code policies. It requires developing wrappers for every system resource that catch calls to such resource and check authorization policies. Strata Security API [33] where systems are hosted on a strata virtual machine which enables interception of system execution at instruction level based on user security policies. The framework does not support securing distributed systems and it focuses on low level policies specified in C code.

Serenity [32] enables provisioning of appropriate security and dependability mechanisms for ambient/intelligence systems at runtime. Security attributes are specified on system components at design time. At runtime the framework links such Serenity-aware systems to the appropriate security and dependability patterns. Serenity does not support dynamic or runtime adaptation for new unanticipated security requirements.

Morin et al. [26] propose a security-driven and model-based dynamic adaptation approach enabling adapting applications to reflect defined context-aware access control (AC) policies. Engineers define security policies that take

into consideration context information. Whenever the system context changes, the proposed approach updates the system architecture to enforce the suitable security policies. Mouelhi et [27] introduce a model-driven security engineering approach to specify and enforce system access control policies at design time based on AOP-static weaving. These adaptive approaches require design time preparation (to manually write integration code or to use specific platform or architecture). They support limited security objectives such as AC. Unanticipated security requirements are not supported. No validation that the target system (after adaptation) correctly enforces security as specified.

3.3 Multi-tenancy security Engineering

The area of multi-tenant SaaS applications' security is relatively new. Possible solutions to multi-tenancy security are still under development by both industry and academia. Michael et al [6] discuss the limitations of security solutions proposed by different commercial cloud platforms. Salesforce [1] has introduced a simplified solution to support their CRM integration with tenants' security solutions. They focus on the Identity and Access Management (IAM) security content only. Tenants who are interested in integrating with Salesforce have to implement web services with a predefined signature.

Enabling applications to support multi-tenancy either during application development or by adapting existing web applications to support multi-tenancy has been investigated by [9, 23, 36, 39]. Cai et al [8, 7] propose an approach to transform existing web applications into multi-tenant SaaS applications. They focus on the isolation problem by analysing applications and identifying the required isolation points that should be handled by the application developers. Guo et al [12] developed a multi-tenancy enabling framework. The framework supports a set of common services that provide security isolation, performance isolation, etc. Their security isolation pattern considers the case of different security requirements of different tenants while still using a predefined, built-in, security controls. It depends on the tenants administration staff to manually configure security policies and map their users and roles to the application predefined roles.

Pervez et al [30] developed a SaaS architecture that supports multi-tenancy, security and load dissemination. The architecture is based on a set of services that provide routing, logging, security. Their proposed security service delivers predefined authentication and authorization mechanisms. No control by service consumers of the security mechanisms is supported and no isolation is provided between the authentication and authorization of data of different tenants. Xu et al [38] proposed a new hierarchical access control model for the SaaS model. Their model adds higher levels to the access control policy hierarchy to be able to capture new roles such as service providers' administrators (super and regional) and tenants' administrators. Service provider administra-

tors delegate the authorization to the tenants' administrators to grant access rights to their corresponding resources.

Zhong et al. [40] propose a framework that tackles the trust problem between service consumers, service providers and cloud providers on being able to inspect or modify data under processing in memory. Their framework delivers a trusted execution environment based on encrypting and decrypting data before and after processing inside the execution environment while protecting the computing module from being access from outside the execution environment. Menzel et al [22] propose a model-driven platform to compose services that represent the SaaS application. Their approach focuses on enabling cloud consumers to compose their system instances while defining their security requirements to be enforced on the composed web services. These security requirements are transformed into WS-policies-alike, applied on the resulting application, and deployed on a separate VM. The limitation of this approach is that it depends on building separate instances for each consumer. There is no means to update or reconfigure the defined security requirements or to extend the enforced security using third party security controls. These efforts we have surveyed deliver security using specific solutions and architectures.

3.4 Industrial Security Platforms

Existing industrial platforms including the Java Security Model, Spring Security Framework (acegi), and Microsoft Windows Identify Foundations (WIF), all help in securing systems by providing a set of security functions and mechanisms. However they usually require developers involvement in writing integration code with such platforms. Thus the resultant systems are tightly coupled with these platforms' capabilities and mechanisms. In addition, using third-party controls requires updating the system source code. Compared to existing efforts, MDSE@R does not assume specific system architecture or a security platform; no security code is required and no developers' involvement in integrating security controls; provides a comprehensive and extensible security model; and third-party security controls can be easily integrated with the target system.

4 MDSE@R

Our model-driven security engineering at runtime (MDSE@R) approach enables service tenants to be involved in securing their cloud hosted assets. As a consequence of tenants' involvement, a single service need to support capturing and enforcing different sets of security requirements of different tenants that become known at runtime. We name this as "Tenant-oriented Security" compared to the traditional security-oriented security where a service instance

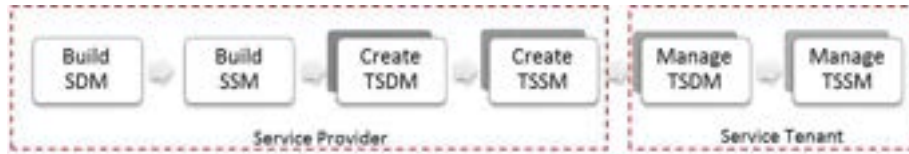


Fig. 2 Process flow of MDSE@R

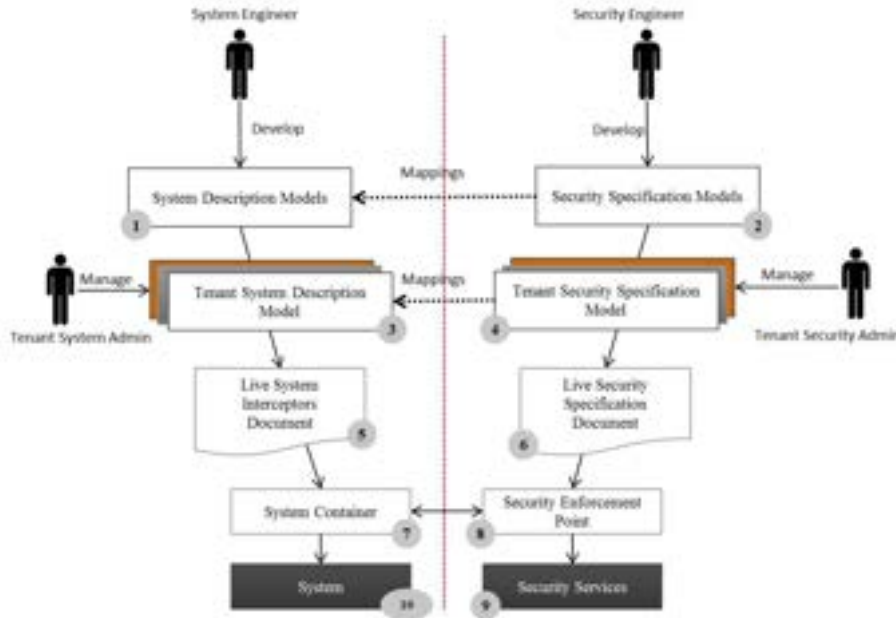


Fig. 3 Overview of MDSE@R approach

reflects only one set of security controls captured by the service provider at design time. Tenant-oriented security may require integrating cloud services with security controls selected by tenants and deployed on or out of the cloud platform. We base our MDSE@R on two key concepts: (i) externalizing security management and enforcement tasks from the application to be secured while being able to wrap the application and intercept calls to any arbitrary critical application entity at runtime using dynamic weaving AOP; and (ii) Model-Driven Engineering (MDE), using Domain-Specific Visual Language (DSVL) models to capture application and security attributes at different levels of abstraction. We automate the generation of security controls integration code rather than using hand-coding of bespoke solutions.

Fig. 2 shows the basic flow of MDSE@R to support multi-tenant security engineering. Service providers develop a detailed service description model (SDM). Then, they develop a security specification model (SSM) capturing all security details they deliver in their cloud service. Once a service tenant

registers to use the service, they will get a copy of the service SDM and SSM. Tenants can then use these models to manage (updated, delete, add) their instances and develop their security needs. Tenants can modify their security requirements and MDSE@R will then automatically update the enforced security on the running application to meet the tenant's new security requirements.

Fig. 3 gives an overview (covering main artifacts developed, key stakeholders, and key interactions) of the MDSE@R approach to support multi-tenant security engineering and tenants' security management at runtime. After capturing application and security models, the MDSE@R platform realizes such modeled changes using interceptors and AOP approach that injects security handlers into the target (secured) application entities (components, classes, and methods) as follows.

4.1 Modeling Service and Security Details

In this phase, stakeholders, from both the service providers and tenants, develop different models that capture details of the service, tenant instance, service security, and tenants' security details as follows:

4.1.1 Build Service Description Model (SDM)

A detailed service description model is delivered by the service provider (an example is shown in Fig. 7). This SDM captures various details of the target application including system features (using use case diagrams), system architecture (using component diagrams), system classes (using class diagrams), system behavior (using sequence diagrams), and system deployment (using deployment diagrams). These models cover most of the perspectives that may be required in securing a given system. Not all these models are mandatory. Tenant security engineers may need to specify security on system entities (using system components and/or classes models), on system status (using system behavior model), on hosting nodes (using system deployment model), or on external system interactions (using system context model). They may specify their security requirements on a coarse-grained level (using system features and components models), or on a fine-grained (using system class diagrams). The service SDMs can be synchronized with the running instance using models@runtime synchronization techniques [11], or manually by the service provider (models@runtime (reflection) research efforts enable management of consistency between system models and running software instances at runtime by reflecting any software changes to system models and vice versa). This also helps in supporting dynamic adaptation i.e. how to develop adaptive systems where system updates are applied on system models and then realized on system instances.

Some of the application or service description details, specifically the system class diagrams, can be reverse-engineered, if not available, from the target

application. We developed a new UML profile to extend UML models with attributes that help in: (i) capturing relations between different system entities in different models e.g. a feature entity in a feature model with its related components in the component model and a component entity with its related classes in the class diagram; and (ii) capturing security concepts/attributes (requirements, controls, etc.) mapped to the SDM entities e.g. security requirements specified on a given system feature or component. This helps in security enforcement and models weaving as we discuss later.

4.1.2 Build Service Security Specification Model (SSM)

A set of models developed and managed by the service provider security engineers to specify the security requirements/controls that the service providers enforce on their services (an example is shown in Fig. 8). It covers the details required during the security engineering process including: security goals and objectives, security risks and threats, security requirements, security architecture for the operational environment, and security controls to be enforced. These models capture different levels of abstractions. The key mandatory model in the security specification models set is the security controls model. It is required for generating the security integration required code (as described below).

4.1.3 Manage Tenant Service Description Model (TSDM)

This model describes system features, architecture and classes available for tenant T. It is usually different from one tenant to another. It depends on the multi-tenancy model adopted by the service provider in customizing tenant instance or configuring a single shared service instance. At tenant provisioning time, an initial TSDM is created as a copy from the system SDM. Then TSDM is updated to reflect current tenant's service instance details. Tenant system administrator can use this model later to turn features on/off at runtime. The TSDM helps in two scenarios: (i) to customize or configure the system based on tenant requirements e.g. tenant T is permitted to use certain features that he registered for. The service provider uses the tenant initial TSDM and delete other system features that are not required. The same approach can be used in both cases either the tenant has a separate instance or share the same instance with other tenants. Still the model@runtime synchronization techniques can be used to keep the TSDM synchronized with the running tenant instance; and (ii) to capture tenants' security requirements on their instance scope i.e. their features, components, methods, etc.

4.1.4 Manage Tenant Security Specification Model (TSSM)

This model is the tenant copy of the service SSM. It describes security objectives, requirements, architecture, design, and controls that the service tenants

have and want to enforce on their cloud-hosted assets. This may include authentication, authorization, auditing, encryption controls that tenants use in their internal sites or even from other security vendors. Tenants may decide to continue using the same security provided by the service provider or rather prefer to use their security controls. However, tenants will not be able to disable security controls that the service provider has marked as mandatory in the service SSM. This helps to avoid disabling critical security controls such as tenants' data isolation control provided by the service provider. This model can be used by tenants to manage security of multiple cloud-hosted applications i.e. single security model to manage all enterprise outsourced assets.

4.2 Weaving Service and Security Models into a Secure-Service Model

MDSE@R has two mapping levels. First, mapping SSM entities to service SDM entities. This mapping is developed and managed by the service providers at design time, deployment time, or even at runtime. Whenever the service provider discovers a security problem or has a new security requirement, they can directly apply it on the service security specification model and then map it to the service description model. Such a mapping is directly reflected on the tenants' models. Second, mapping TSSM entities to service TSDM entities. This mapping is developed and managed by the service tenant at runtime. Both mappings can be modified at runtime to reflect new needs.

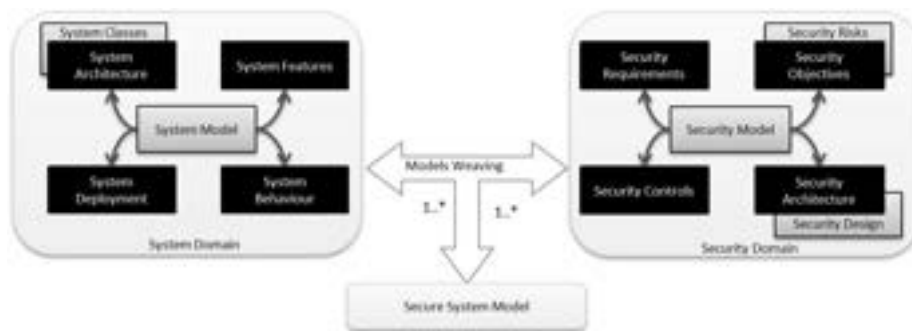


Fig. 4 Possible Weavings of service and security models

MDSE@R supports many-to-many mapping between the (tenant) service description model entities and (tenant) security specification model (SSM) entities, as shown in Fig.4. This is supported by our UML profile which extends every service description concept i.e. feature, component, class, method, host, connection, etc. with a set of security attributes i.e. security objectives, requirements, services and controls (see Fig. 11 for UML profile diagram). Using drag-and-drop between the SSM and SDM entities, SSM entity will be added as an attribute value, based on the dragged SSM entity, to the selected SDM

entity. One or more security entities (security objective, requirement and/or control) can be mapped to one or more service model entity (feature, component, class or method). Mapping a security concept on an abstract service entity e.g. a system feature - implies a delegation of the same security concept to the concrete entities e.g. the feature realization classes and methods. This is facilitated using our UML profile, which helps in managing traceability between application entities. Mapping an abstract security concept e.g. a security objective to a service entity - e.g. a class - implies mapping all security requirements, services, and controls that realize this security objective to that class. Any application entity that has a security mapping is called a critical application entity.

4.3 Enforcing Specified Security on Target Application Entities

In the previous steps, both security details and critical application entities emerge at runtime. MDSE@R automates the realization of the specified security on the critical application entities without the involvement of security or application engineers. This helps both parties to easily update their application and security capabilities to meet their needs. This helps in avoiding inconsistency problems that usually arise from the need to maintain both models and realizations, which force administrators and developers to update the security realizations directly.

Whenever the service provider or service tenant develops a new mapping or updates an existing mapping between an SSM entity and an SDM entity, the underlying MDSE@R platform propagates these changes as follows:

- Update Live Service Interceptors' Document (Fig.3-5). This document maintains the list of critical application entities (CP -an application entity that has security attributes mapped on it) where security controls should be weaved or integrated. Equation 1 states that the critical service entities - CP(s) - are the union of all tenants' critical points - CP (Ti) where To is the service provider.

$$CP(s) = \bigcup_{i=1}^{i=n} CP(T_i) \quad (1)$$

- Update Live Security Specification Document (Fig.3-6). This document maintains the list of security controls to be applied at every critical system entity. This may be defined by the service provider or by the service tenant. In this case we have to mark the service provider security controls as they should be enforced/applied first before any request to this critical system entity.
- Update the System Container (Fig.3-7). The container is responsible for intercepting system calls to critical system entities at runtime and delegating such requests to the default handler, the "Security Enforcement Point. Update Tenant Accessible Resources Document. This document maintains a list of system resources that should not be accessible for each tenant e.g.

if Swinburne did not buy the Customer Management module, they should not be able to access webpages or functionalities provided in this module. Equation 2 is used in specifying the tenant's inaccessible resources. The prohibited resources list for tenant T_i is the difference between the service SDM resources and the tenant T_i TSDM resources.

$$PR(T_i) = R(SDM) - R(TSDM(T_i)) \quad (2)$$

This list of tenant's prohibited resources is used by the MDSE@R to deny access to any of them for any given request submitted by one of the tenant's users. The application or service is now ready to enforce security specified by tenants and service providers based on the woven secure-service model. This update is conducted in parallel with the application or service operation. Thus it does not incur any further performance overhead.

4.4 Security Services

A key objective of MDSE@R is to avoid being tightly coupled with specific security controls, specific vendor, or specific security platform (Java security manager, spring acegi framework, Microsoft Windows Identity Foundations, etc.). in addition to keeping developers and administrators away from being deeply involved in integrating security controls with a target system which usually result in inconsistent security being enforced on different systems. We developed a common security interface for every security attribute (authentication, authorization). This interface, shown in Fig.5, specifies functions and signatures that each security control expects/requires in order to perform their tasks e.g. user identity, credentials, roles, permissions, claims, etc. A security control or service vendor must implement this interface in their connector or adapter to support integration with MDSE@R. This helps security vendors develop one connector that – using MDSE@R – can be integrated with all target applications/services.

4.5 Security Enforcement Point - SEP

So far we have prepared the system to intercept requests to critical methods via the system container and have prepared security controls to be communicated using the common security interface. The Security Enforcement Point (SEP) works as a bridge between the system container and the deployed security controls. SEP queries the security specification document for controls to enforce at every intercepted request. It then initiates calls (using the common security interface) to the designated security controls' clients or connectors. The SEP assigns results returned by such controls to the system context e.g. an authentication control returns userID of the requesting user after being authenticated. The SEP creates an Identity object from this userID and assigns it to the current thread' user identity attribute. Thus a secured application

```

1  public interface SecurityStandardInterfaceAuthentication
2  {
3      //Authentication
4      public string[] AuthenticateUser();
5      public bool IsAuthenticated(object subject);
6  }
7  public interface SecurityStandardInterfaceAuthorization
8  {
9      //Authorization
10     public string[] AuthorizeUser(object subject);
11     public bool IsAuthorized(object subject, object action,
12                             object resource, string context);
13 }
14
15 public interface SecurityStandardInterfaceCryptography
16 {
17     //Cryptography
18     public string Hash(string plaintext);
19     public string Encrypt(string plaintext);
20     public string Decrypt(string ciphertext);
21     public string DigitalSign(string data);
22     public bool VerifySignature(string signature, string data);
23 }
24
25 public interface SecurityStandardInterfaceEncoding
26 {
27     //Encoding
28     public string Encode(string input);
29     public void AddCSRFToken();
30 }
31
32 public interface SecurityStandardInterfaceLogging
33 {
34     //Logging
35     public void Log(int type, string message);
36 }
37
38 public interface SecurityStandardInterfaceInputValidation
39 {
40     //Input Validation
41     public bool IsValid(string ruleName, string input)
42 }

```

Fig. 5 MDSE@R simplified common security interface

can work normally as if it has authenticated the user by itself. An application may use such information in its operations e.g. to insert a record in the DB, it uses the user identity to set the "enteredBy" DB field.

4.6 Testing the System-Security Integration

Before allowing the developed specifications and mappings to be applied to the live application or service, MDSE@R uses a security testing service to verify that the target application is now correctly enforcing the specified security

needs. We assume that security controls are already tested by the security vendors. Thus, our testing task should focus on verifying that security controls are correctly integrated within specified critical system entities. To automate this step, we use the live interceptors' document and the security specification document to generate a set of test cases (scripts) for each critical entity i.e. each critical entity will have a set of test cases according to the security specified on it (from the security specification document). Each test case verifies that a security control C is correctly integrated within the critical entity E. To test the correct integration, we simulate requests to E and check the resultant system security context (after calls - actual results) against the expected results e.g. user identity is correctly set, permissions are set as specified, etc. Finally, we generate a log of the test cases firing results to the tenant/service provider security engineers showing the failed test cases, the critical entities, and the failed to integrate security controls.

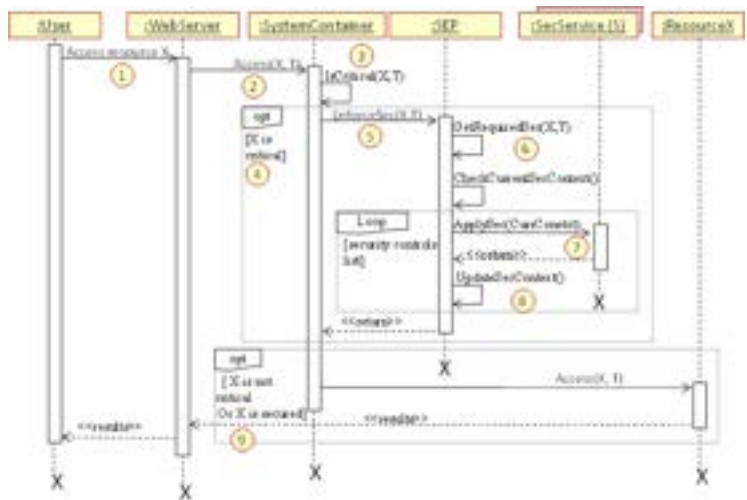


Fig. 6 Sequence diagram of a user request to critical application entity

Fig. 6 shows a sequence diagram describing a user requesting resource X, from a service operated by MDSE@R. In this scenario we have several interacting entities including user, webserver, MDSE@R system container, SEP, security services and the application resource. Once the web server receives a request submitted by the user to access resource X (1), it delegates the request to the system container along with the user's tenantID T (2). The system container queries the live service interceptors' document (3) to decide if the resource requested is marked as "critical" or not either by the tenant or by the service provider. If the resource is critical (4), the system container delegates the request to the security enforcement point (5). The SEP queries the security specification document for the required security controls to be enforced

on the requested resource by the user's tenant or by the service provider (6). This is an ordered list of security controls to be activated by the SEP. The SEP loops on the retrieved security controls list. Using the common security interface, the SEP generates requests to security controls required according to their type/family (7). After each security control call, the SEP updates the current threat security context (8). Finally, the SEP returns to the system container a recommendation to either proceed with the request or to deny it (9). If appropriate, the system container then forwards the request to the target resource or else returns an appropriate security exception to the caller.

5 Usage Example

The key objective of this usage example is to show how the service provider and tenants can collaborate together using MDSE@R and our platform toolsets to manage security of their services at runtime. We highlight the key stakeholders involved in the security engineering process along with their responsibilities and their expected outcomes of every step. We use the motivating example from Section 2, Galactic application developed by SwinSoft and procured by Swinburne and Auckland. SwinSoft wants to adapt its application security at runtime to block security holes and vulnerabilities that have been discovered at runtime. Two tenants using Galactic are worried about the security of their assets and have their own, different security requirements to be enforced on their Galactic ERP application instance(s). The examples of models in the Figures are taken from our prototype DSVL tool in use for this scenario.

5.1 Model Galactic System Description

This task is done during or after the system is developed. SwinSoft, the service provider, decides the level of application details to provide to their tenants in Galactic SDM. Fig. 7-A shows that SwinSoft SDM captures the description of system features including customer, employee and order management features (Fig. 7), system architecture details including Presentation Layer, Business Logic Layer (BLL) and Data Access Layer (DAL) (Fig. 7-B), system classes including CustomerBLL, OrderBLL, EmployeeBLL (Fig. 7-C), and system deployment including web server, application server, and data access server (Fig. 7-D). SwinSoft uses our UML profile (Fig. 13) to capture dependencies and relationships between system features and components, and system components and classes. This model is used as a reference by SwinSoft system and security engineers. No tenant is allowed to have write access to the Galactic SDM or its details.

5.2 Model SwinSoft Security Details

This task is conducted by SwinSoft (service provider) security engineers at the system deployment phase. This model is usually updated during their repetitive security management process to reflect new risks. In this scenario, SwinSoft security engineers document SwinSoft security objectives that must be satisfied by Galactic system (Fig. 8-A) including data integrity with medium importance, confidentiality with high importance, accountability with low importance. This model should be repeatedly revised to incorporate emerging changes in SwinSoft security objectives. Security engineers then refine these security objectives in terms of security requirements that must be implemented by the Galactic system, developing a security requirements model. A part of it is shown in Fig. 8-B including authentication requirements. This model keeps track of the security requirements and their links back to the high level security objectives. In this example, we show that the `AuthenticateUser` requirement is to be enforced on Galactic along with its detailed sub-requirements.

SwinSoft security engineers next develop a detailed security architecture including services and security mechanisms to be used in securing Galactic (Fig. 8-C). In this example, we show the different security zones (big boxes - Fig. 8-C) that cover SwinSoft network and the allocation of IT systems, including Galactic. The security architecture also shows the security services, security mechanisms and standards that should be deployed. SwinSoft security engineers finally specify the security controls (i.e. the real implementations) for the security services modelled in the security architecture model (Fig. 8-D). This includes `SwinValidator`, `ESAPI.AccessController`, and `SecurityIsolator` security controls. Each security control entity defined in the security controls model specifies its family (authentication, authorization, audit, etc.) and the deployment URL of its connector. Each security specification model maintains traceability information to parent model entities. In Fig. 8-d, we specify that `SecurityIsolator` realizes the "TenantsDataIsolation" requirement. Whenever MDSE@R finds a system entity with a mapped security requirement `TenantsDataIsolation` it adds `SecurityIsolator` as its realization control i.e. an `SecurityIsolator` check will run before the entity is accessed e.g. before a method is called or a module loaded. SwinSoft security engineers have to mark mandatory security controls that their tenants cannot modify or disable.

Fig. 7 Examples of the Galactic system description model

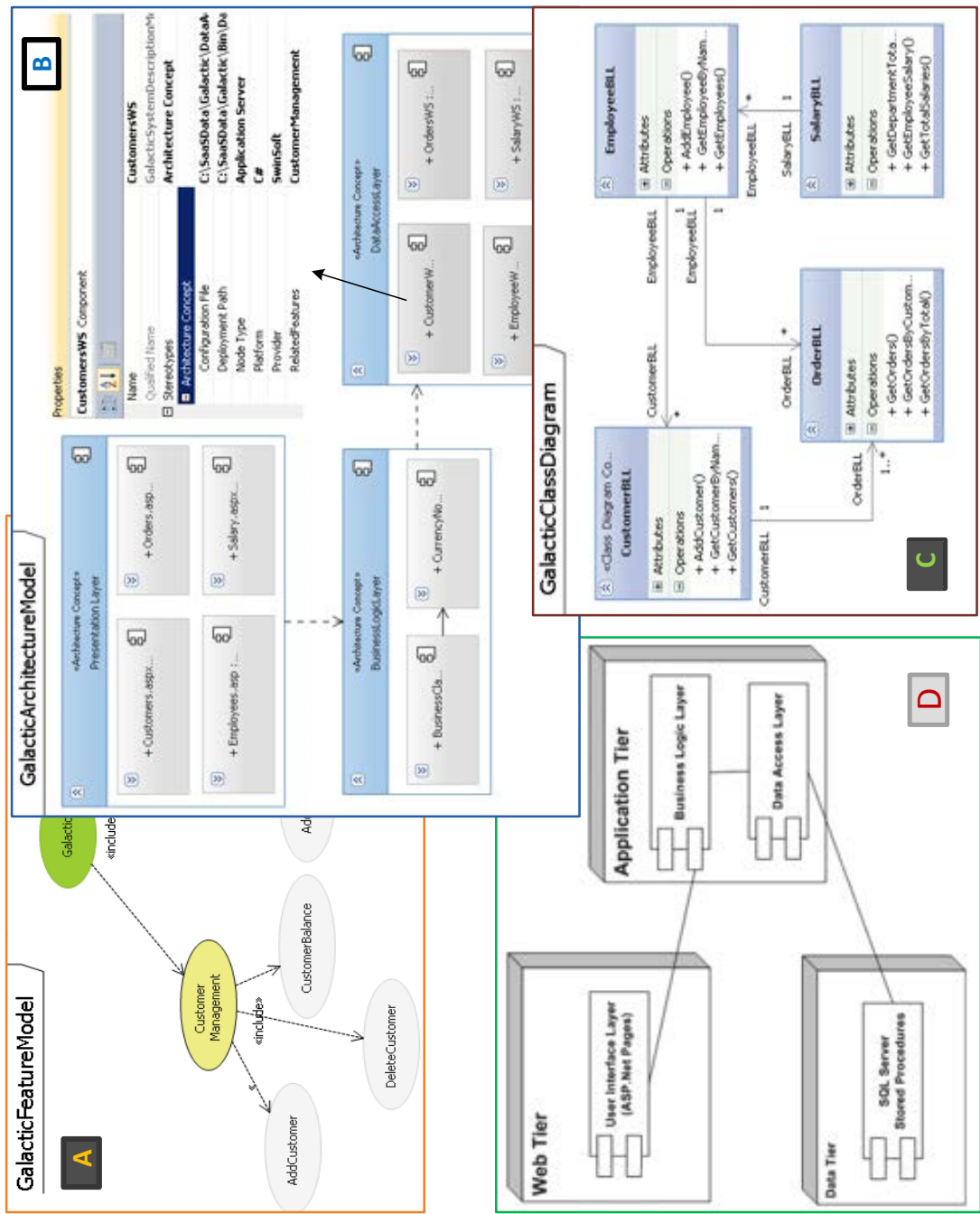
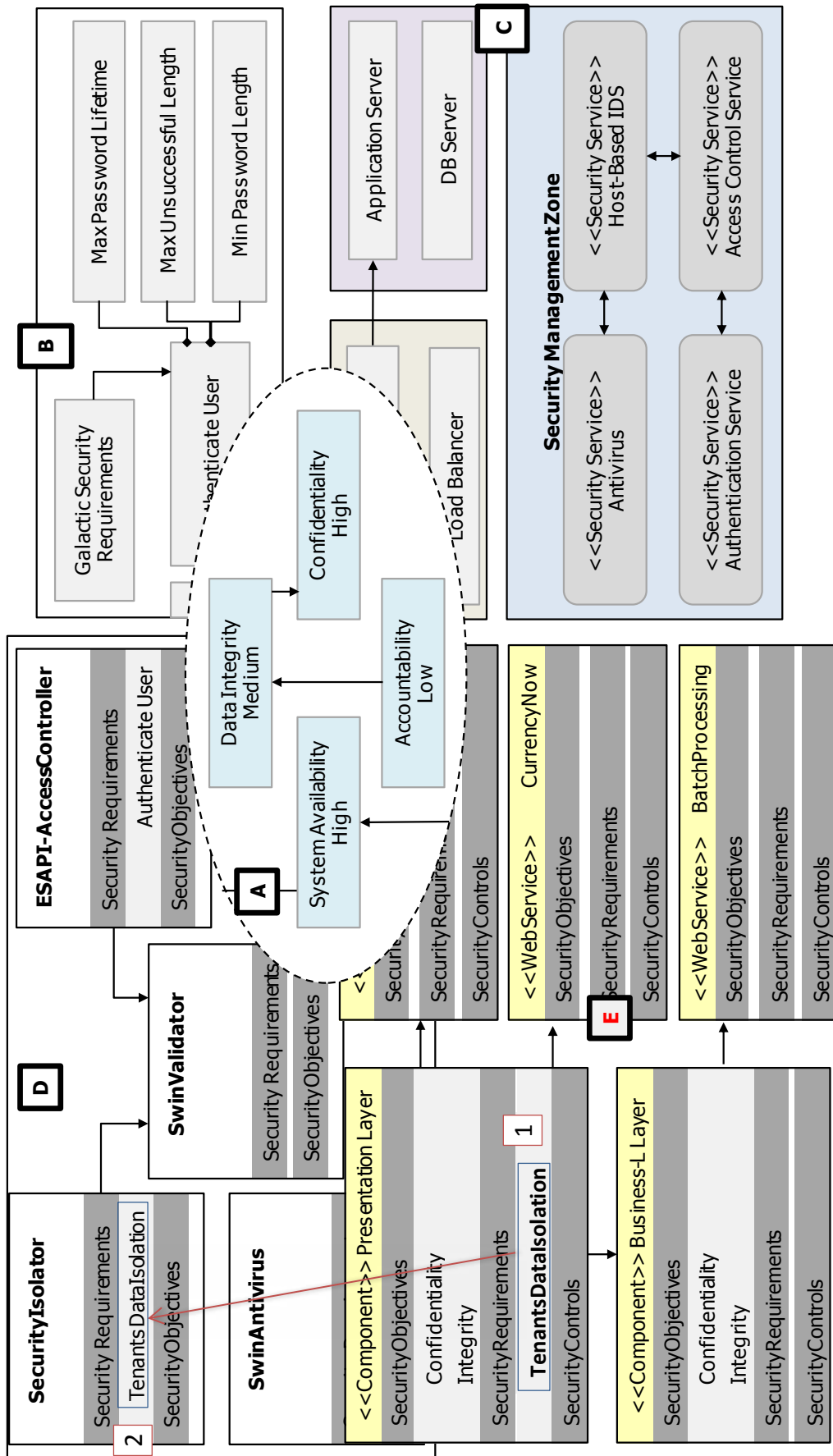


Fig. 8 Examples of SwinSoft Security Specification Models



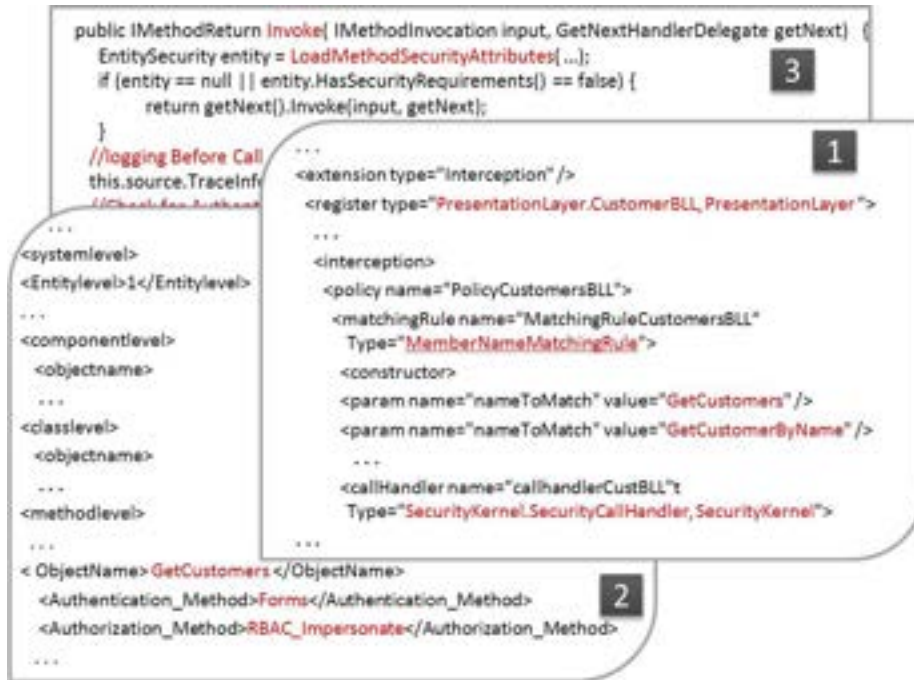


Fig. 9 Examples of the interceptors and security specification files

5.3 Weave System SDM and Security SSM

After the development of the Galactic SDMs and the security SSMs by SwinSoft security engineers, the SwinSoft security engineers map security attributes (in terms of objectives, requirements and controls) to Galactic system specification details (in terms of features, components, classes). This is achieved by drag and drop of security attributes to system entities. Thus, system feature, structure, or behaviour can dynamically and at runtime reflect different levels of security based on the currently mapped security attributes on it. Fig. 8-E shows a part of Galactic component diagram where `PresentationLayer`, a UML component entity, is extended with security objectives, requirements and controls compartments. In this example the security engineers have specified `TenantsDataIsolation` as one of the security requirement to be enforced on the `PresentationLayer` component (1). Such a requirement is achieved indirectly using `SecurityIsolator` control (2). MDSE@R uses the security attributes mapped to system entities to generate the full set of interceptors for system method calls, as in Fig. 9-1 (system interceptors document), and application entities' required security controls, as in Fig. 9.2 (security specification document).

```
[Test]
public void AuthorizeGetCustomersTesting() {
    CustomerBLL obj = new CustomerBLL();
    GenericPrincipal testIdentity =
        new GenericPrincipal(new GenericIdentity("TestIdentity"),
            new string[] { "TestRole" });

    try {
        obj.GetCustomers();
    }
    catch(Exception ex) {
        Assert.Fail("Firing of AuthorizeGetCustomersTesting test case failed");
    }
    Assert.AreNotEqual(testIdentity, System.Threading.Thread.CurrentPrincipal,
        "LDAP Authorization Control is not correctly plugged-in", null);
}
```

Fig. 10 MDSE@R samples of the generated security integration test cases

5.4 Testing Galactic Security

Once security has been specified and interceptors and configurations generated, MDSE@R verifies that the system is correctly enforcing security as specified. MDSE@R generates and fires a set of required security integration test cases. Our test case generator uses the system interceptors and security specification documents to generate a set of unit test cases for each method listed in the interception document. The live systems interceptor document represents the source of system points we need to test for security integration. The live security specification document represents the source of security controls that we need to test their integration with the critical system points. The MDSE@R testing service has a set of predefined security control unit test templates for each security control attributes e.g. authentication, authorization, input validation, etc., including tests for successful and unsuccessful security enforcement. These unit test templates are used to generate test cases by replacing tags with specific security control names and critical point names. These generated tests are then run against the application and the results inspected to determine pass/failure. An example of a generated test case is shown in Fig10. This contains a set of security assertions (one for each security attribute specified on a given system entity). During the firing phase, the security enforcement point is instrumented with logging transactions to reflect the calling method, called security control, and the returned values. Security engineers should check the security test cases firing log to verify that no errors introduced during the security controls integration with Galactic entities. After SwinSoft security engineers have checked the MDSE@R Security testing service log files and make sure that no integration errors have been introduced, they can publish their updated Galactic security model for their tenants.

5.5 On-boarding Swinburne and Auckland Tenants

During tenants on-boarding process (preparing the service to be used by the new tenant), SwinSoft system engineers/admins start to customize/configure Galactic instance for tenant based on their requirements and purchased modules. Depending on the adopted multi-tenancy model, they may register new features or components as well. The final Swinburne or Auckland TSDM looks like the Galactic SDM in Fig.7. Swinburne and Auckland system administrators can update their own tenants TSDMs to reflect any further system customization, such as enabling or disabling sub-features such as calculate overtime, nightshifts, and vacations in the Employee Management module. This TSDM is used by the Swinburne security engineers to define required security on it. The updates done by SwinSoft or Swinburne on the TSDM are reflected on the prohibited resources list (Eq.2)

5.6 Swinburne and Auckland Manage their TSDMs and TSSM

Swinburne and Auckland security engineers go through the same process as SwinSoft did when specifying their security requirements and controls. Each tenant can customize their TSSM as far as they want and as frequent as required. For example, in Fig. 11 Swinburne engineers have specified that LDAP "realizes the AuthenticateUser requirement. Whenever MDSE@R finds a system entity with a mapped security requirement AuthenticateUser it adds LDAP as its realization control i.e. an LDAP authentication check will run before the entity is accessed - e.g. before a method is called or a module loaded. This applies to the CustomerBLL class methods, Fig11-(1). However, Swinburne security engineers have a different requirement for the GetCustomers method - the requester should be authenticated using Forms-based authentication as well, Fig11-(2). Auckland can specify their specific requirements, context, and security controls based on their specific needs. This results in quite different generated security enforcement controls. Both Swinburne and Auckland security engineers can modify the security specifications while their Galactic application is in use. MDSE@R framework updates interceptors in the target systems and enforces changes to the security specification for each system as required. For example, the Swinburne Galactic security model can be updated with a Shibboleth single sign-on ¹security authentication component and these updates applied to the running Galactic deployment.

6 MDSE@R Platform Architecture and Implementation

The architecture of MDSE@R platform is shown in Fig. 12. This is designed to support managing multiple SaaS applications hosted on a cloud platform. MDSE@R consists of system and security specification modellers, models and

¹ <http://shibboleth.net/>

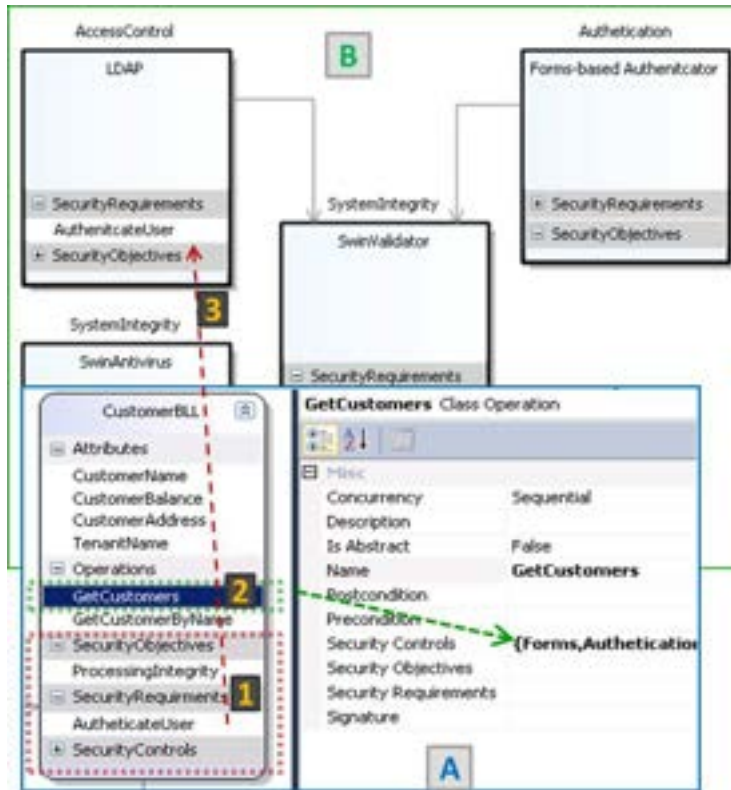


Fig. 11 Examples of Swinburne Security Specification Models

security controls specifications repositories, system container to intercept request, testing services and security enforcement point.

Our System Description Modeller (1) was developed as an extension of Microsoft Visual Studio 2010 modeller with an UML profile (Fig. 13) to enable system engineers modelling their systems' details with different perspectives including system features, components, deployment, and classes. The UML profile, as shown in Fig. 13, defines stereotypes and attributes to maintain the track back and forward relations between entities from different models. A set of security attributes to maintain the security concepts (objectives, requirements and controls) mapped to every system entity (Fig.7). The minimum level of details expected from the system provider is the system deployment model. MDSE@R can use this model to reverse engineer system classes and methods using .NET Reflection (in case of system binaries only available). We use .NET parsers to extract classes and methods from the system source code by analysing the generated Abstract Syntax Tree (AST) files by .NET parsers.

Our Security Specification Modeler (2) was developed as a Microsoft Visual Studio 2010 plug-in. It enables service providers and tenants, represented

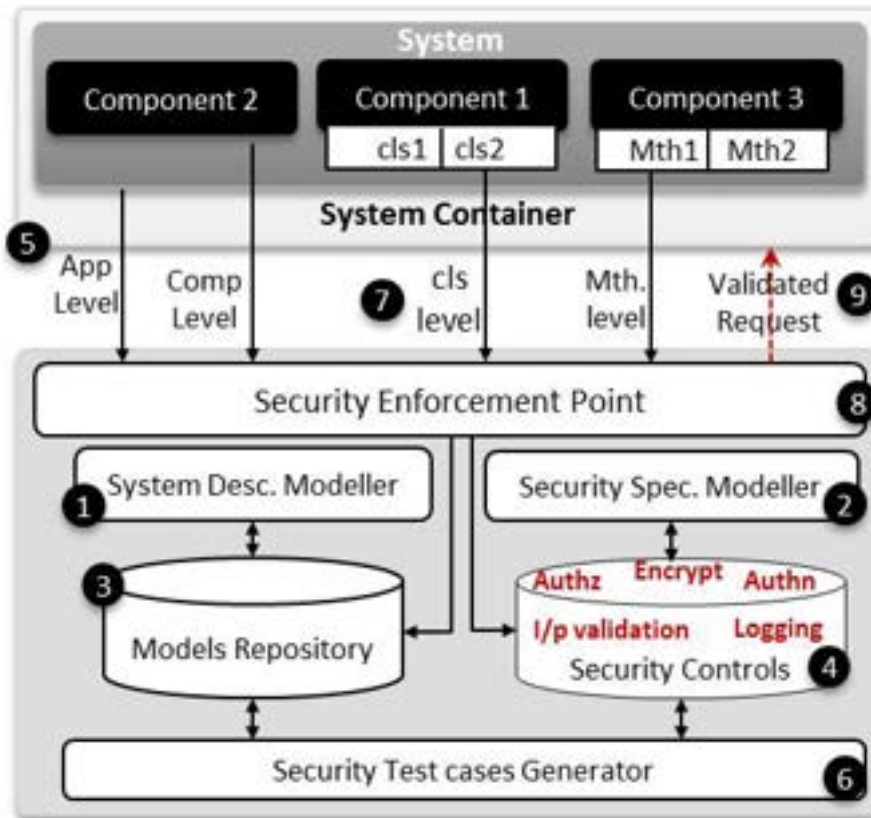


Fig. 12 MDSE@R platform architecture

by their security engineers, to specify the security attributes and capabilities that must be enforced on the service and/or its operational environment. The security modeler delivers a set of complete security DSVLs. Fig. 14 shows the meta-model of the MDSE@R security DSVL². The security-objectives DSL captures customer's security objectives and the relationships between them. Each objective has a criticality level and the defence strategy to be followed: preventive, detective or recovery. The Security requirements DSL captures customer's security requirements and relationships between requirements including composition and referencing relations. The Security Architecture DSL captures security architectures and designs of the customer operational environment in terms of security zones and security level for each zone; security objectives, requirements and controls to be enforced in each zone; components and systems to be hosted in each zone; security services, mechanisms and standards to be deployed in each zone or referenced from other zones. The Security Controls DSL captures details of security controls that are registered and de-

² <http://www.ict.swin.edu.au/personal/malmorsy/Pubs/TR002.pdf>

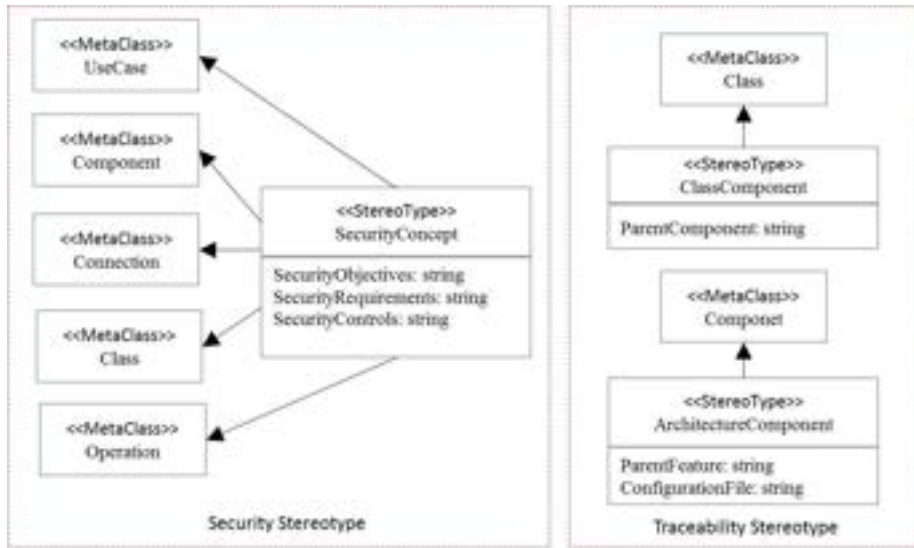


Fig. 13 MDSE@R system description model UML profile

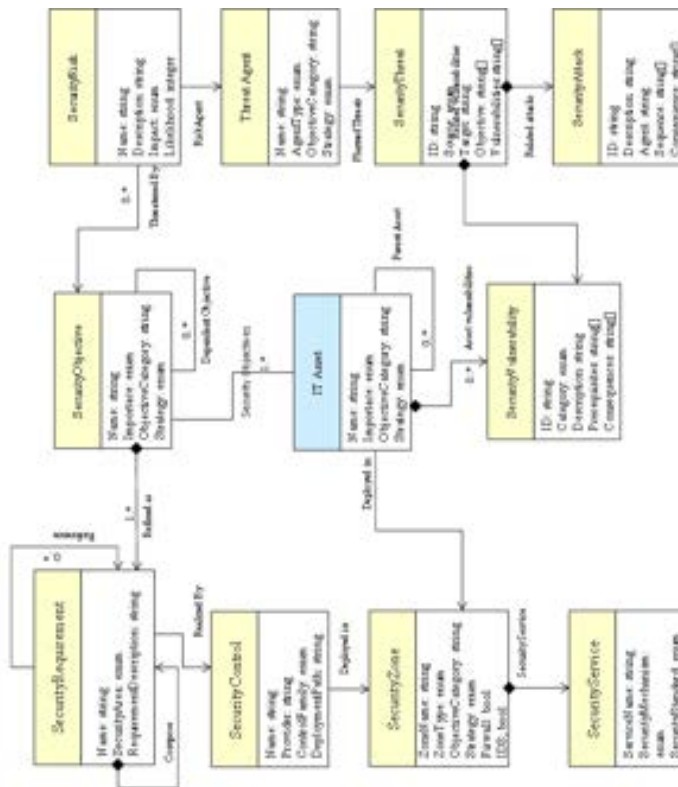


Fig. 14 MDSE@R security meta-model

ployed in the customer environment and relationships between these and the security requirements they cover.

Models Repository (3): Both modeling tools use a shared repository to maintain models developed either by the system engineers or the security engineers. This repository also maintains the live system interceptors' document and security specification document. An example of these documents is shown in Fig 9. This example shows a sample of the Galactic interceptors document generated from the specified security-system mapping. It informs the system container to intercept `GetCstomerByName` and `GetCustomers` methods (1); a sample of Swinburne security specification file defining the security controls to be enforced on every intercepted point (2); and a sample of the security enforcement point API that injects the necessary security control calls before and after application code is run (3).

Security Controls Database (4) is a database of the available and registered security patterns and controls. It can be extended by the service providers or by a third party security provider. A security control must implement certain APIs defined by the security enforcement point in order to be able to integrate with the target system security standard interface. Having a single enforcement point with a predefined security interface for each security controls family enables security providers to integrate with systems without having to redevelop adopters for every system. We adopted OWASP Enterprise Security API (ESAPI) library³ as our security controls database. It provides a set of authentication, authorization, encryption, etc. controls that we used in testing our approach.

System Container (5): To support run-time security enforcement, MDSE@R uses a combined dependency injection and dynamic-weaving AOP approach. Whenever a client or application component sends a request to any critical system component method, this request is intercepted by the system container. The system container supports wrapping of both new developments and existing systems. For new development, Swinsoft system engineers should use the Unity application block delivered by Microsoft PnP team⁴ to support intercepting any arbitrary class entity. Unity supports dynamic runtime injection of interceptors on methods, attributes and class constructors. For existing systems we adopted Yiihaw AOP [31], where we can modify application binaries (dll and exe files) to add security aspects at any arbitrary system method (we add a call to our security enforcement point). For component level interception, we can use `httpModules` to add our interceptor on the component level.

Our Security Test Case Generator (6) uses the NUnit testing framework⁵ to partially automate security controls and system integration testing. We developed a test case generator library that generates a set of security test cases for authentication, authorization, input validation, and cryptography for

³ <https://www.owasp.org/index.php>

⁴ <http://pnp.azurewebsites.net/en-us/>

⁵ <http://www.nunit.org/>

every enforcement point defined in the interceptors document. MDSE@R uses NUnit library to fire the generated test cases and notifies security engineers via test case execution result logs.

At runtime, whenever a request for a system resource is received (7), the system container checks for the requested method in the live interceptors' document. If a matching found, the system container delegates this request with the given parameters to the security enforcement point (8). Security Enforcement Point (9) is a class library that is developed to act as the default interception handler and the mediator between the system and the security controls. Whenever a request for a target application operation is received, it checks the system security specification document to enforce the particular system security controls required. It then invokes such security controls through APIs published in the security control database (4). The security enforcement point validates a request via the appropriate security control(s) specified, e.g. imposes authentication, authorization, encryption or decryption of message contents. The validated request is then propagated to the target system method for execution (10).

7 Evaluation

In this section we summarize some of the experiments we have performed to assess the capabilities and scalability of our MDSE@R approach in:

- Capturing descriptions of different real systems and different security details for both service providers and tenants;
- Propagating security attributes on different system entities (features, components, classes, and methods);
- Enforcing unanticipated security requirements including authentication, authorization, auditing, etc. at runtime with an acceptable performance overhead;
- Validating that security controls are correctly integrated with the target entities.

7.1 Benchmark applications setup

We have selected a set of seven web-based, real-world, large, widely-used, and commercial open source web applications developed using ASP.Net (currently we have a .NET parser only) as our benchmark to evaluate MDSE@R. These applications cover a wide business spectrum. We divided the evaluation set into two groups: Group-1 (G-1) has two applications including Galactic (an ERP system developed internally in our group for testing purposes) and PetShop (a well-known reference e-Commerce application). Both applications have been modified to adopt the Unity application block as the system container. Group-2 (G-2) has five third-party web applications. SplendidCRM is an open source CRM that is developed to with the same capabilities of the well-known open

source SugarCRM system. It has been downloaded more than 400 times. SugarCRM has a commercial and community version. KOOBOO is an open source enterprise CMS used in developing websites. It has been downloaded more than 2000 times. BlogEngine is an open source ASP.NET 4.0 blogging engine. It has been downloaded more than 46000 times. BugTracer is an open-source, web-based bug tracking and general purpose issue tracking application. It has been downloaded more than 500 times. NopCommerce is an open-source eCommerce solution. It has more than 10 releases. For this group we use Yiihaw framework as the system container to inject interceptors into system binaries. Except for Galactic, we do not have any previous experience with these applications. Table 1 shows some statistics about the selected set of applications including Lines of codes in (KLOC), No. of files, No. of components, No. of classes, and No. of methods included. It is clear that benchmark applications vary from large-scale systems such as SplendidCRM, KOOBOO, and NopCommerce to small scale such as PetShop.

Table 1 Benchmark applications statistics

Benchmark	KLOC	Files	Components	Classes	Methods
Galactic	16.2	99	7	101	473
PetShop	7.8	15	5	25	256
SplendidCRM	245	816	28	6177	6107
KOOBOO	112	1178	34	7851	5083
NopCommerce	442	3781	45	5127	9110
BlogEngine	25.7	151	4	258	616
BugTracer	10	19	2	298	223

Table 2 Security controls used by service provider, Swinburne, Auckland

Sec. Attribute	SwinSoft Ctls	Swinburne Ctls	Auckland Ctls
Authn.	ESAPI	Forms-based	LDAP
Authz.	ESAPI	Forms-based	LDAP
I/P santization	ESAPI	-	-
Audit	ESAPI	PrivateAuditor	PrivateAuditor
Cryptography	ESAPI	DES	AES
Sec. Isolation	ESAPI	-	-

7.2 Experimental setup

Using MDSE@R, we developed three security specification models (SSM) with security objectives, requirements, and controls as in Fig. 7. One model for ser-

vice provider and two other models were copied from it and modified to reflect two security requirements sets. We specified security requirements and controls for authentication, authorization, input validation, logging and cryptography as shown in Table 2. We used MDSE@R to model the system description (SDMs) for applications in Group-1, as we know the details of these systems. For Group-2, we used system deployment diagram for these applications and used MDSE@R to reverse engineer systems' class diagram from there binaries. Thus for applications in Group-1 we should be able to map security to system features, components, classes, methods. However, in Group-2 we should be able to specify security on component, class, and method levels only.

7.3 Evaluation results

Table 3 shows security attributes that MDSE@R succeeds in capturing and enforcing at runtime, including authentication, authorization, input sanitization, auditing and cryptography. This represents most common security attributes. Table 3 also shows that MSDE@R succeeded in mapping and enforcing these security attributes on all systems in both Group-1 and Group-2 with different levels of system abstractions (F: feature, C: component, S: class, and M: method). Note that for Group-2 applications we do not have a system feature model to map and enforce security on this level. The enforcement of cryptography has a limitation with Group-2 applications especially when securing methods. This is because it requires that the caller and callee expect parameters of type String. To address this problem, we used format-preserving encryption (FPE) techniques. The output of these techniques is in the same format (type) of the input - i.e. if the input to encrypt is of type integer then the output is of the same type.

Table 3 Results of validating MDSE@R against Group-1 and Group-2 applications

Benchmark App.		Security Attributes					
		Sec. Isolation	Authn.	Authz.	Input Sanitization	Audit	Cryptography
Group-1	Galactic	F, C, S, M	F, C, S, M	F, C, S, M	F, C, S, M	F, C, S, M	F, C, S, M
	PetShop	F, C, S, M	F, C, S, M	F, C, S, M	F, C, S, M	F, C, S, M	F, C, S, M
Group-2	Splendid	C, S, M	C, S, M	C, S, M	C, S, M	C, S, M	(C, S, M)*
	KOOBOO	C, S, M	C, S, M	C, S, M	C, S, M	C, S, M	(C, S, M)*
	NopCommerce	C, S, M	C, S, M	C, S, M	C, S, M	C, S, M	(C, S, M)*
	BlogEngine	C, S, M	C, S, M	C, S, M	C, S, M	C, S, M	(C, S, M)*
	BugTracer	C, S, M	C, S, M	C, S, M	C, S, M	C, S, M	(C, S, M)*

7.4 User evaluation

We carried out a preliminary user evaluation of our tools and platform to assess MDSE@R approach and platform usability. We had seven post-graduate researchers, not involved in the development of the approach, use our developed tools and platform after receiving an hour training session on the tool

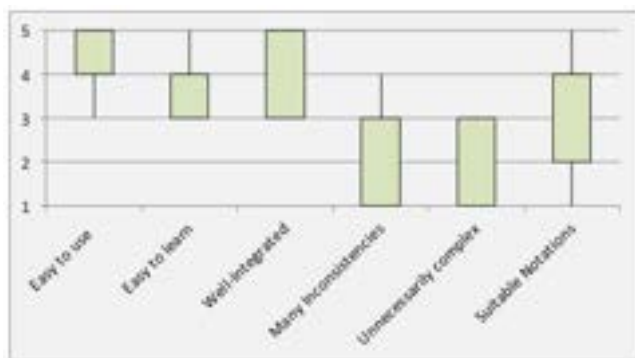


Fig. 15 Level of agreement of usability factors (1: Strongly disagree ... 5: Strongly agree)

and platform features. We asked them to explore several MDSE@R system and security DSVL specifications of the PetShop and Galactic applications. Then we asked them to perform updates on these models and to modify the security specification models at run-time. We conducted a basic usability survey to gain their feedback on our DSVL, modelling tools, and the security enforcement platform. The results show that they successfully understood and updated security models for the target systems. They gave positive feedback about the overall approach and the tool usability, and the capabilities in managing system security, as shown in Fig. 15 (1: Strongly disagree to 5: Strongly agree). A key recommendation was to use more expressive icons in the security DSVL rather than just boxes.

7.5 Performance evaluation

In this section, we discuss the performance evaluation of our MDSE@R platform in both runtime performance overhead and offline adaptation overhead.

7.5.1 Runtime performance overhead

The runtime performance overhead of MDSE@R equals time incurred intercepting requests, plus time spent by the security enforcement module in querying the security requirements repository to be enforced on the intercepted point, plus time spent in calling the security controls specified. Time spent by the security controls themselves we do not factor in, as this needs to be spent whether using our approach or traditional hard-coded security solutions. Arguably, traditional approaches may incur some of these other time penalties as well e.g. checking authenticated user access controls or generating audit checkpoint information to log. Fig. 16 shows the time required (in msec) by MDSE@R to process a request for systems with different numbers of concurrent users and different number of system entities that have been marked as

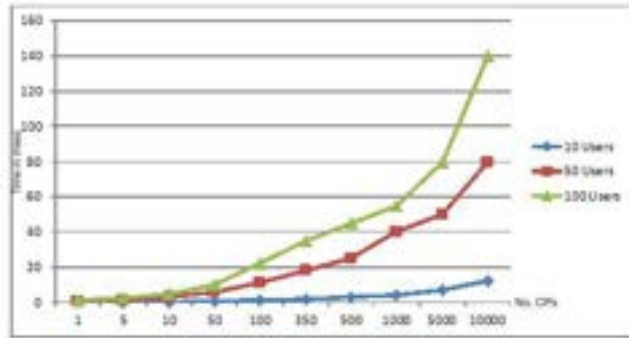


Fig. 16 Average performance overhead of MDSE@R platform

critical. Experiments were conducted on a Core2Duo desktop PC with 4GB Memory. The max performance overhead we got for a system with 10000 CPUs defined and having 100 users concurrently sending requests equals 140msec. This performance considers efficient memory utilization as interceptors and security specification documents are loaded as needed. Significantly better performance could probably be achieved by caching these MDSE@R models in memory and using a hash table data structure to enable faster search. Using replicas of the MDSE@R platform on different servers and for different applications will result in further improvement of its performance overhead.

7.5.2 Security adaptation overhead

We have measured the adaptation delay incurred by MDSE@R in order to realize a single simple mapping between a security entity and a system entity e.g. system methods. This overhead equals on average 3 seconds. This represents the time taken to update the security interceptors and security specification documents and time to generate and fire the required integration test case(s). This is an offline task and so does not impact the performance of the system running instance.

8 Discussion

Our MDSE@R approach promotes multi-tenancy security engineering from design time to runtime. This is based on externalizing security engineering activities including capturing objectives, requirements controls, and realization from the target system implementation. This permits both security to be enforced and critical system entities to be secured to evolve at runtime (supporting adaptive security at runtime). It enables enforcing different security requirements sets for different tenants who are not known at design time. We name this as "Tenant-oriented security" compared to the traditional service-oriented security where a service can reflect only one set of security

requirements usually captured by service provider at design time. Finally, a key benefit reaped from MDSE@R approach is to the support model-based security management. Tenant security requirements, architecture and controls are maintained and enforced through a set of centralized TSSMs instead of low level scattered configurations and code that lack consistency and are difficult to modify. A tenant can have a single TSSM for all of their IT systems that captures all of their security specifications and can be updated anytime to reflect his new configurations. Thus any update to their TSSM will be reflected on all IT systems that use MDSE@R platform.

In our evaluation we developed one security model and used it with different systems. Each system enforces the security mapped to its entities. Any update to the security model results in updating all systems linked to it. This is a key issue in environments where multiple applications must enforce the same security requirements. Having one place to manage security reduces the probability of errors, delays, and inconsistencies. Automating the propagation of security changes to underlying systems simplifies the enterprise security management process. The multi-tenancy security engineering of existing services (extending system security capabilities) has three possible scenarios: (i) for systems that already have their SDMs, we can use MDSE@R directly to specify and enforce multi-tenant security at runtime; (ii) for systems without SDMs, we can reverse engineer parts of the required system models (specifically the class diagram) using MDSE@R (if these binaries can be read and not obfuscated). Then we can use MDSE@R to engineer required system security; (iii) for systems with built-in security, we can use MDSE@R to add new security capabilities only. MDSE@R cannot itself help modifying or disabling existing security. However, we have been working on extending our approach to support deletion of existing security methods and partial code using modified AOP techniques [2]

The selection of the level of details to apply security on depends on the criticality of the system. In some situations, we may intercept calls to the presentation layer only (webserver) while considering the other layers secured by default (not publicly accessible). In other cases, such as integration with a certain web service or using third party component, we may need to have security enforced at the method level (for certain methods only). Security and performance trade-off is another dilemma to consider. The more security validations and checks the more resources required. This impacts application performance. This should be included as a part of the Service Level Agreement (SLA) with the tenants. We plan to extend our generated test cases to include performance tests in the near future, allowing MDSE@R provider to assess the overhead of new security configurations in terms of cost and to help both providers and tenants to optimize the security level enforced. MDSE@R helps in engineering security into systems at runtime, while the security controls configuration and administration should be managed by security administrators. MDSE@R does not support defining business rules at runtime e.g. an employee should not be able to retrieve a customer's records if customer is of type VIP. The target

system should have this rule while MDSE@R will provide the current user roles/permissions as returned by the tenant security control.

Security isolation between different tenants' data is a very critical requirement in engineering security of a multi-tenant SaaS application. In MDSE@R, we consider security isolation as one of the security controls that simply performs authorization of the tenants supplied inputs before proceeding with the requests. Thus no tenant can access other tenants data by providing malicious inputs. However, the service providers have to perform the data filtration when loading/storing data from/to the application database. One may argue that our approach may lead to a more open and vulnerable system as we did not consider security engineering during design time. Our argument is that at design time security engineering is often done by security non-experts and this is a key reason why we still discover a lot of vulnerabilities in deployed systems. However, service providers can still perform security engineering during design time using MDSE@R. The service provider delivers both the SDM and SSM to their tenants for further customization. This also helps small tenants or tenants who are satisfied with the delivered security.

9 Summary

MDSE@R is a new model-driven approach to dynamically engineering security for multi-tenant SaaS applications at runtime. Our approach is based on using a set of multi-level service description models (SDM), developed by service providers, to describe different perspectives of their applications; a set of security specification models (SSM), developed by the service provider, to capture security objectives, requirements and environment security controls using Domain-Specific Visual Languages. Then, tenants can customize their copies of the SDM and SSM to reflect their application and security configurations. MDSE@R then bridges the gap between these two specifications through merging of the service and security models for both service provider and service tenants into a joint service security model.

MDSE@R uses dynamic injection of security enforcement interceptors and code into the target application to enforce the security specified. Security specifications are thus externalized and loosely coupled with application specifications, enabling both the application and security specification to evolve. It also allows sharing of security specification models among different applications "model-based security management". Security controls can be integrated with MDSE@R (which was implicitly integrated with the tenant service) by implementing a common security interface that we have introduced.

We have developed a set of modeling tools and a prototype of MDSE@R. We have successfully validated our approach by applying it to seven web-based applications, most of them open source, successfully modeling and enforcing a range of security needs on these applications. We performed a preliminary user evaluation of our toolset that demonstrates that it is readily usable by a technical audience but with little security engineering background. We assessed

the performance overheads of using our current prototype of MDSE@R. It has a performance overhead ranging from 0.13msec up to 140msec per request for each critical application entity. MDSE@R has adaptation delay of 3sec for each simple mapping between SSM and SDM. This represents time to update interceptors and security specification documents as well as generating and firing security test cases.

Acknowledgements Funding provided for this research by Swinburne University of Technology and FRST SPPI project is gratefully acknowledged. We also thank Swinburne University of Technology for their scholarship support for the first and third authors.

References

1. AKAI, S., AND CHIBA, S. Extending aspectj for separating regions. ACM, 2009.
2. ALMORSY, M., GRUNDY, J., AND IBRAHIM, A. S. Supporting automated software re-engineering using re-aspects. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 230–233.
3. ALMORSY, M., GRUNDY, J., AND MUELLER, I. An analysis of the cloud computing security problem. In *Prof. of 2010 Asia Pacific Cloud Workshop, Colocated with APSEC* (Sydney, Australia, 2010).
4. ANDERSON, R. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.
5. BAUER, A., AND JUERJENS, J. Security protocols, properties, and their monitoring. In *Proceedings of the fourth international workshop on Software engineering for secure systems* (New York, NY, USA, 2008), SESS '08, ACM, pp. 33–40.
6. BROCK, M., AND GOSCINSKI, A. Toward a framework for cloud security algorithms and architectures for parallel processing. vol. 6082 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 254–263.
7. CAI, H., WANG, N., AND ZHOU, M. J. A transparent approach of enabling saas multi-tenancy in the cloud. In *Services (SERVICES-1), 2010 6th World Congress on* (5-10 July 2010 2010), pp. 40–47.
8. CAI, H., ZHANG, K., ZHOU, M. J., GONG, W., CAI, J. J., AND MAO, X. S. An end-to-end methodology and toolkit for fine granularity saas-ization. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on* (21-25 Sept. 2009 2009), pp. 101–108.
9. CHINCHANI, R., IYER, A., NGO, H., AND UPADHYAYA, S. A target-centric formal model for insider threat and more. Tech. rep., Technical Report 2004-16, University of Buffalo, US, 2004.
10. ELKHODARY, A., AND WHITTLE, J. A survey of approaches to adaptive application security. In *Int. Workshop on Software Engineering for Adaptive and Self-Managing Systems* (2007), pp. 1–16.

11. GORDON BLAIR, NELLY BENCOMO, R. B. F. Models@run.time. In *IEEE Computer* (2009), pp. 22–27.
12. GUO, C. J., SUN, W., HUANG, Y., WANG, Z. H., AND GAO, B. A framework for native multi-tenancy application development and management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on* (23-26 July 2007 2007), pp. 551–558.
13. HAFNER, M., MEMON, M., AND BREU, R. Seaas - a reference architecture for security services in soa. *Journal of Universal Computer Science vol. 15* (2009), 2916–2936.
14. HASHII, B., MALABARBA, S., PANDEY, R., AND AL, E. Supporting reconfigurable security policies for mobile programs. North-Holland Publishing Co., 2000.
15. JURJENS, J. Towards development of secure systems using umlsec. In *Fundamental Approaches to Software Engineering*, vol. 2029. Springer Berlin Heidelberg, 2001, ch. Lecture Notes in Computer Science, pp. 187–200.
16. JURJENS, J., AND WIMMEL, G. Formally testing fail-safety of electronic purse protocols. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on* (nov. 2001), pp. 408 – 411.
17. LAMSWEERDE, A., BROHEZ, S., AND AL, E. System goals to intruder anti-goals: Attack generation and resolution for security requirements engineering. In *Proc. of the 3rd Workshop on Requirements for High Assurance Systems* (Monterey, 2003), ACM, pp. 49–56.
18. LIU, L., YU, E., AND MYLOPOULOS, J. Secure i* : Engineering secure software systems through social analysis. *International Journal of Software and Informatics Vol.3*, pp. 89-120 (2009).
19. LODDERSTEDT, T., B. D., AND DOSER, J. Secureuml: A uml-based modeling language for model-driven security. In *The 5th International Conference on The Unified Modeling Language* (Dresden, Germany, 2002), vol. 2460, Springer-Verlag, pp. 426–441.
20. MEAD, N., AND STEHNEY, T. Security quality requirements engineering (square) methodology. ACM, 2005.
21. MELLADO, D., FERNNDEZ-MEDINA, E., AND PIATTINI, M. Applying a security requirements engineering process. In *Computer Security ESORICS 2006*, D. Gollmann, J. Meier, and A. Sabelfeld, Eds., vol. 4189 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 192–206.
22. MENZEL, M., WARSCHOFSKY, R., THOMAS, I., AND WILLEMS, C. MEINEL, C. The service security lab: A model-driven platform to compose and explore service security in the cloud. In *Services (SERVICES-1), 2010 6th World Congress on* (5-10 July 2010 2010), pp. 115–122.
23. MIETZNER R., LEYMANN F., P. M. P. Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *Internet and Web Applications and Services*,

2008. *ICIW '08. Third International Conference on* (8-13 June 2008 2008), pp. 156–161.
24. MONTRIEUX, L., JÜRJENS, J., HALEY, C. B., YU, Y., SCHOBENS, P.-Y., AND TOUSSAINT, H. Tool support for code generation from a umlsec property. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 357–358.
 25. MORIN, B., BARAIS, O., NAIN, G., AND AL, E. Taming dynamically adaptive systems using models and aspects. In *IEEE 31st Int. Conf. on Software Engineering* (Vancouver, BC, 2009), IEEE Computer Society, pp. 122–132.
 26. MORIN, B., MOUELHI, T., FLEUREY, F., AND AL, E. Security-driven model-based dynamic adaptation. ACM, 2010.
 27. MOUELHI, T., FLEUREY, F., BAUDRY, B., AND ET AL. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 11th Int. Conf. on Model Driven Engineering Languages and Systems* (France, 2008), Springer-Verlag.
 28. MOURATIDIS, H., AND GIORGINI, P. Secure tropos: A security-oriented extension of the tropos methodology. *International Journal of Software Engineering and knowledge Engineering* (2007).
 29. PERVEZ, Z., LEE, S., AND LEE, Y.-K. Multi-tenant, secure, load disseminated saas architecture. In *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on* (7-10 Feb. 2010 2010), vol. 1, pp. 214–219.
 30. PERVEZ, Z., LEE, S., AND LEE, Y.-K. Multi-tenant, secure, load disseminated saas architecture. In *Proceedings of the 12th international conference on Advanced communication technology* (Gangwon-Do, South Korea, 2010), IEEE Press, pp. 214–219.
 31. RASMUS JOHANSEN, STEPHAN SPANGENBERG, P. S. Yiihaw .net aspect weaver usage guide.
 32. SANCHEZ-CID, F., AND MANA, A. Serenity pattern-based software development life-cycle. In *19th International Workshop on Database and Expert Systems Application* (2008), pp. 305–309.
 33. SCOTT, K., KUMAR, N., VELUSAMY, S., AND AL, E. Retargetable and reconfigurable software dynamic translation. IEEE Computer Society, 2003.
 34. SINDRE, G., AND OPDAHL, A. Eliciting security requirements with misuse cases. *Requir. Eng.* 10, 1 (2005), 34–44.
 35. THOMAS VOGEL, ANDREAS SEIBEL, H. G. The role of models and megamodels at runtime. In *Proceedings of the 2010 international conference on Models in software engineering* (2010), pp. 224–238.
 36. WANG, D., ZHANG, Y., ZHANG, B., AND LIU, Y. Research and implementation of a new saas service execution mechanism with multi-tenancy support. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering* (2009), IEEE Computer Society, pp. 336–339.
 37. XIN JIN, RAM KRISHNAN, R. S. A unified attribute-based access control

- model covering dac, mac and rbac. In *Proceedings of the 26th Annual IFIP WG 11.3 conference on Data and Applications Security and Privacy* (2012), pp. 41–55.
38. XU, J., JINGLEI, T., DONGJIAN, H., LINSEN, Z., LIN, C., AND FANG, N. Research and implementation on access control of management-type saas. In *2010 The 2nd IEEE International Conference on Information Management and Engineering (ICIME)* (16-18 April 2010 2010), pp. 388–392.
 39. ZHANG, X., SHEN, B., TANG, X., AND CHEN, W. From isolated tenancy hosted application to multi-tenancy: Toward a systematic migration method for web application. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on* (16-18 July 2010 2010), pp. 209–212.
 40. ZHONG, C., ZHANG, J., XIA, Y., AND YU, H. Construction of a trusted saas platform. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE)* (4-5 June 2010 2010), pp. 244–251.

Mohamed Almorsy received his Bachelor and Masters degrees in computer science, Ainshams University, Cairo, Egypt. Mohamed has more than seven years of experience in the software development industry. He began his PhD at Swinburne University of Technology in 2010. Mohameds main focus is cloud computing security management.

John Grundy is Professor of Software Engineering. He is the Head of Computer Science & Software Engineering and Director of Centre for Complex Software Systems and Services Centre Director, Centre for Computing and Engineering Software Systems, Faculty of Information & Communication Technologies, Swinburne University of Technology.

Amani S. Ibrahim received his Bachelor and Masters degrees in computer science, Ainshams University, Cairo, Egypt. Amani began her PhD at Swinburne University of Technology in 2010. Amani focuses mainly on securing the cloud computing infrastructure using virtualization-aware security solutions.

3.4 SoftArch: tool support for integrated software architecture development

Grundy, J.C., and Hosking, J.G. SoftArch: tool support for integrated software architecture development, *International Journal of Software Engineering and Knowledge Engineering*, vol 13, no 2, April 2003, World Scientific, pp. 125-151.

DOI: [10.1142/S0218194003001238](https://doi.org/10.1142/S0218194003001238)

Abstract: A good software architecture design is crucial in successfully realising an object-oriented analysis (OOA) specification with an object-oriented design (OOD) model that meets the specification's functional and non-functional requirements. Most CASE tools and software architecture design notations do not adequately support software architecture modelling and analysis, integration with OOA and OOD methods and tools, and high-level, dynamic architectural visualisations of running systems. We describe SoftArch, an environment that provides flexible software architecture modelling using a concept of successive refinement and an extensible architecture meta-model. SoftArch provides extensible analysis tools enabling developers to analyse their architecture model properties. Run-time visualisation of systems uses dynamic annotation and animation of high-level architectural modelling views. SoftArch is integrated with a component-based CASE tool and run-time monitoring tool, and has facilities for 3rd party tool integration through a common exchange format. This paper discusses the motivation for SoftArch, its modelling, analysis and dynamic visualisation capabilities, and its integration with various analysis, design and implementation tools.

My contribution: Developed initial ideas for this research, co-designed approach, wrote the software the approach based on, wrote majority of the paper, co-lead investigator for funding for this project from FRST

SOFTARCH: TOOL SUPPORT FOR INTEGRATED SOFTWARE ARCHITECTURE DEVELOPMENT

JOHN GRUNDY^{1,2} JOHN HOSKING¹

*Department of Computer Science¹ and Department of Electrical and Electronic Engineering²,
University of Auckland
Private Bag 92019, Auckland, New Zealand
john-g@cs.auckland.ac.nz*

Submitted 3rd November 2001

First Revision 12th March 2002

Accepted 12th July 2002

Abstract

A good software architecture design is crucial in successfully realising an object-oriented analysis (OOA) specification with an object-oriented design (OOD) model that meets the specification's functional and non-functional requirements. Most CASE tools and software architecture design notations do not adequately support software architecture modelling and analysis, integration with OOA and OOD methods and tools, and high-level, dynamic architectural visualisations of running systems. We describe SoftArch, an environment that provides flexible software architecture modelling using a concept of successive refinement and an extensible architecture meta-model. SoftArch provides extensible analysis tools enabling developers to analyse their architecture model properties. Run-time visualisation of systems uses dynamic annotation and animation of high-level architectural modelling views. SoftArch is integrated with a component-based CASE tool and run-time monitoring tool, and has facilities for 3rd party tool integration through a common exchange format. This paper discusses the motivation for SoftArch, its modelling, analysis and dynamic visualisation capabilities, and its integration with various analysis, design and implementation tools.

Keywords: software architecture, software tools, architecture modelling and analysis, software visualisation

1 Introduction

Many software modelling notations and tools have been developed over time [1, 2, 3, 4]. Due to the increasing complexity of software systems there has been an increasing emphasis on software architecture modelling in CASE tools in addition to the more conventional object-oriented analysis (OOA) and object-oriented design (OOD) modelling [5, 6, 3, 4]. Various design notations have been developed, including those of UML [7], PARSE [4], JViews and aspects [8, 9], tool abstraction [2], and Clock [1, 10]. Support tools include Rational Rose [11], JComposer [9], PARSE-DAT [4], ViTABaL [2], SAAMTool [3] and Argo/UML [12]. The Unified Modelling Language (UML) [7] uses a combination of class, collaboration, component and deployment diagrams. Clockworks and JComposer use annotated component diagrams [1, 9]. PARSE-DAT and ViTABaL use process diagrams. Several systems, including SAAMTool [3], Argo/UML [12] and Visper [13], use various kinds of structural architecture component diagrams.

Most of these systems provide only partial software architecture modelling solutions, supporting some aspects of architecture modelling supported e.g. basic structure, limited dynamic behaviour and event models, or dynamic process creation [15, 16]. Most only capture limited knowledge about an architecture's properties and the characteristics of architecture elements. Few provide analysis tools to help developers reason about their models and ensure OOA requirements are met and architecture components refined to suitable OOD abstractions [12, 15]. Few support OOD and/or implementation code generation from architecture-level abstractions, and few support reuse of previously developed models and patterns [10, 12]. Almost none are sufficiently extensible to allow new architecture abstractions and analysis tools to be added, and most architecture representations in tools have poor or no integration with related analysis, design and

implementation abstractions. High-level dynamic visualisation of algorithms and design-level call graphs and dataflow have been used for many years [17, 18, 19, 14, 20] to provide a mixture of views of running program information. Most of these approaches focus on object or algorithm-level dynamic visualisation techniques, rather than architectural component visualisation. Limited architecture-level visualisations have been developed, together with approaches to visualise running systems [18, 19]. However most dynamic visualisations bear little or no relation to static architecture visualisation (design) notations, making them hard to understand and interpret.

We describe SoftArch, an environment providing new approaches for software architecture modelling, analysis, visualisation and tool integration. Architects use an extensible visual notation to describe and refine software architecture models. Detailed properties of architecture elements and element groupings capture knowledge of architectural characteristics. A collection of extensible “analysis agents” constrain, guide and advise architects as they build and refine these models. Visualisation of running system architectures using high-level abstractions in SoftArch is supported. SoftArch has been integrated with process management, analysis, design and implementation tools, using a variety of tool integration techniques, to “value-add” to a software designer’s overall tool set by providing support for complementary, integrated architecture development.

In the following sections we motivate the need for SoftArch and review current support for architecture modelling, analysis and dynamic visualisation support. We then overview the facilities of SoftArch, focusing in turn on its static architecture modelling, architecture analysis, and dynamic architecture-level visualisation support. We briefly discuss the design and implementation of SoftArch, focusing on its integration with other tools (CASE, programming environments and run-time systems). We conclude with a summary of SoftArch’s contributions and directions for future research.

2 Motivation

Software architecture development has become an increasingly important part of the software lifecycle, due to the increasing complexity of software being constructed [21, 7, 22]. Software developers need to carefully describe and reason about the architectures of complex, distributed information systems, which are often comprised of a mix of new and reused components. A good, extensible and maintainable architecture often makes the difference between successful and failed projects. Much more time tends to be spent on architecture development than previously, and many more options exist for developers [21].

Consider a simple E-commerce system, a screen dump from which is shown in Fig. 1 (a). This is a collaborative travel planning system which provides itinerary views (1), flight bookings (2) and a travel map visualisation (3) [23]. Fig. 1 (b) shows two high-level views of parts of the software architecture for this system from our SoftArch design tool. The top view shows how specific components of the client-side and server-side processes are inter-related. The bottom view shows how customer and travel agent clients access the centralised server processes. These views are described using our SoftArch visual architecture description notation [5, 24, 22]. In order to design and build such a system, developers need to carefully model the software architecture and refine it to a suitable OOD model, ensuring it meets all system functional and non-functional specifications.

We define a software architecture as the organisation of the software elements of a system, together with relationships to the hardware and networking required to run and support communication between these software elements. Like most researchers we characterise software architectures as comprised of various components (groups of functional abstractions) and connectors linking components [5, 22]. Each has both functional (data and behavioural) and non-functional (e.g. performance, reliability, security, integrity, etc) properties. Most OOD techniques, like the UML, focus solely on detailed functional system definition. However, many software architecture description languages (ADLs) aim to associate both functional and non-functional properties of a specification with architectural elements, so these can be reasoned about [21, 5, 24, 16]. An architecture description should help developers to meet a system specification’s functional and non-functional requirements, and a rich variety of architectural views may be useful (data allocation, processes and process inter-connections, subscribe-notify and event-passing approaches, host machines, processes and networking, and so on). Thus when designing the

architecture for a system like the Travel Planner outlined above, developers typically require support to:

- represent processes (clients, servers, databases etc), machines (client and server hosts etc), data and other architectural components (database tables, files, etc) [1, 11]
- represent inter-component relationships, such as structural relationships, data usage, message passing, event subscription/notification, message order, concurrency and so on [16, 4, 12]
- represent additional architectural characteristics related to those above, such as data and control functions, data replication, caching, concurrency control, security mechanisms, communication protocols, etc [1, 2]
- model and reason about both static architectural connections and dynamic behaviour of related architecture components
- capture both functional and non-functional characteristics of each of these architectural features [7, 1, 16]

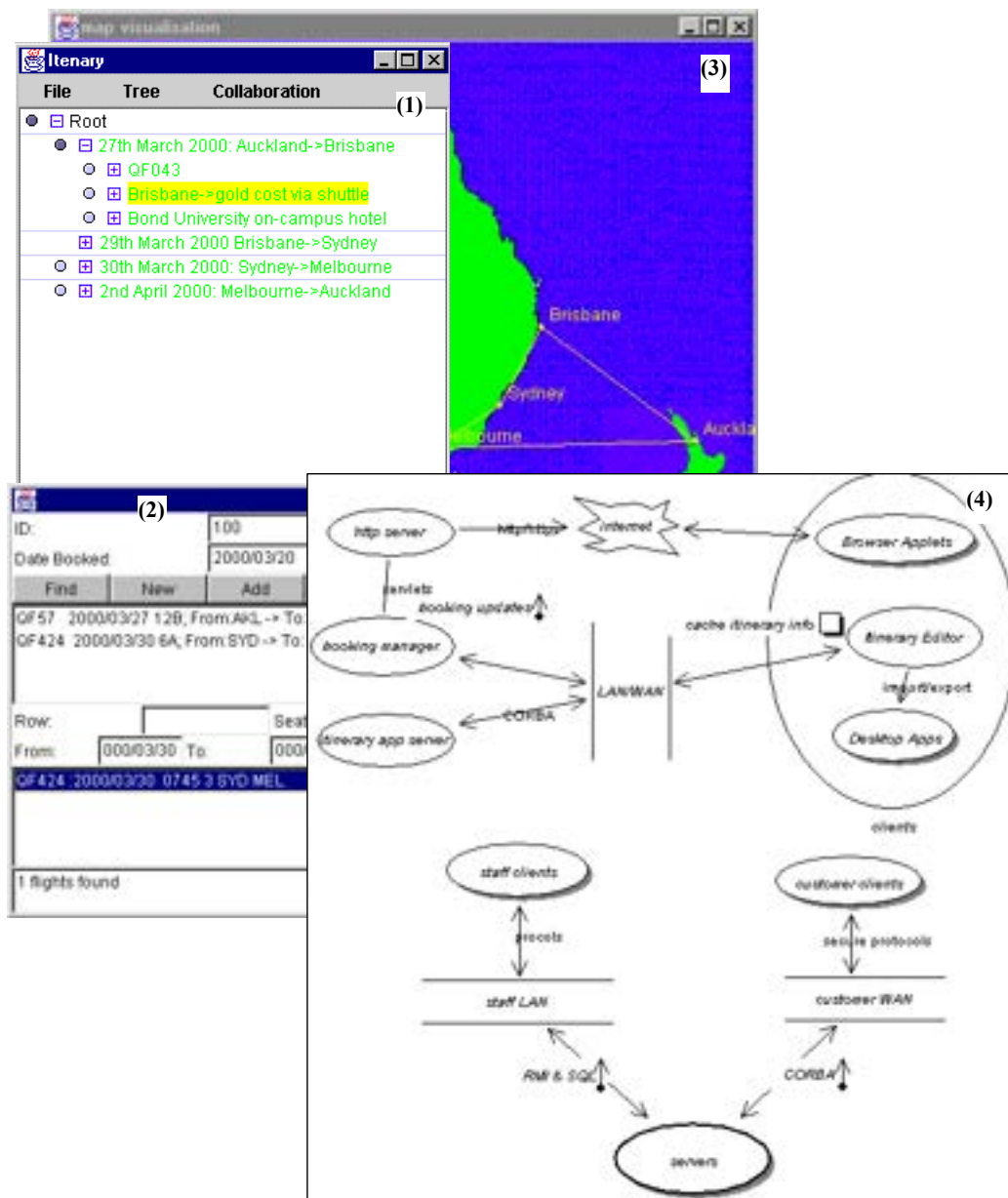


Fig. 1. (a) An example system and (b) two views of parts of its software architecture from the SoftArch environment.

In addition, a system's software architecture can be viewed from many levels of abstraction, from high level (e.g. client-server; staff clients vs customer clients; server processes; multi-tier architecture) to architecture-implementing object-oriented design (OOD) classes and inter-class

relationships (e.g. “TravelItineraryClient”, “FlightManager”, and “CustomerTable”). In any non-trivial architecture there normally exists many refinement steps from OOA specifications and high-level views of the system’s architecture to detailed OOD-level class and object abstractions. Refinements of system functional and non-functional properties, using multiple levels of software architecture abstractions, thus preserves traceability from OOA specifications to low-level OOD design implementation approaches.

Fig. 2 illustrates the development process and relationships between OOA, software architecture, and OOD and implementation-level software artefacts we aim to support with SoftArch. Architects construct architecture designs at varying levels of detail to realise an OOA specification, eventually producing parts of an OO design (to be completed and implemented using e.g. CASE tools and programming environments). In addition, often existing designs and code must be reverse engineered into higher-level architectural models, which themselves may need to be reverse engineered into OOA specifications. Ideally an architecture design tool should support traceability from high-level to low-level architectural abstractions. It should also aid developers in validating the correctness of their use of architectural abstractions. Architectural design views at different levels of abstraction should be able to help developers analyse how an implemented, running system using the architecture behaves.

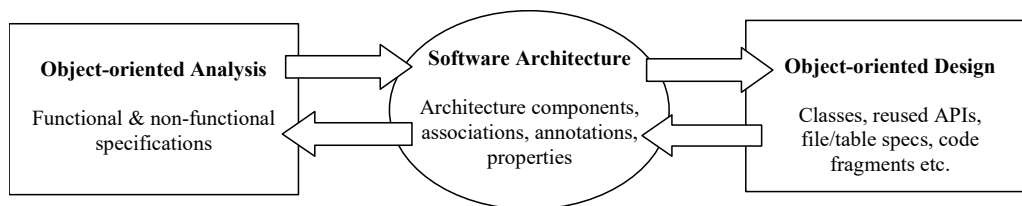


Fig. 2. Transformation of OOA model to OOD model via Software Architecture.

A tool to support architecture modelling, refinement, validation and to utilise static architecture design information to aid running architecture performance analysis should:

- Allow architects to use an extensible set of architecture modelling abstractions i.e. different kinds of components, connectors and component/connector annotations. Architects need to use a wide range of suitable abstractions when designing architectures, and need to extend these for different problem domains. Each of these architecture element types will have a variety of characteristics the designer may specify (e.g. name, location, performance characteristics, required security support, and so on).
- Support modelling the system at differing levels of architectural abstraction, from very high-level to parts of a detailed OO design. Ideally a number of visual abstractions will be provided along with detailed architectural data entry.
- Provide architects with assistance reasoning about and validating complex architecture designs. This should include checking the characteristics of related architecture elements to ensure usage constraints and non-functional properties are consistent and compatible.
- Support visualisation of implemented architecture designs using high-level design abstractions. This allows architects to link implemented system performance results with the architectural abstractions the implementation is based on
- Be able to exchange data with related tools e.g. CASE tools, programming environments, monitoring tools.

3 Related Work

Existing software architecture notations and support tools generally lack comprehensive support for architecture modelling, refinement, analysis and OOA/D linkage [15, 16]. Commonly used software modelling notations like the UML [7] provide views of classes, components and machines. Such notations suit low-level architectural representation reasonably well, but do not provide for higher level architectural oversight [16, 1]. Deployment diagrams in UML offer a view of machine and process assignment and inter-connection, but this is the only high-level specifically architectural view in UML, and is quite limited. Commonly used CASE tools, such as Rational Rose [11] and Argo/UML [12], also lack notational abstractions for designing large system architectures [15]. In addition, most CASE tools lack adequate support for refinement of

OOA to OOD and architecture models and for maintaining traceability between multiple levels of system abstractions. Few provide adequate template or reusable model support and few capture architecture-related design rationale [16].

Most component engineering tools, such as JComposer [9], Borland JBuilder and that of Wagner et al [25], provide little in the way of architecture modelling support, focussing primarily on design- and implementation-level detail. The latter is necessary when developing systems, but too low-level for large system architecture development. Few support capture of multiple perspectives on architecture models and different levels of abstraction and refinement relationships. JComposer [9] and MET+[25] provide component views with some higher level associations and properties like event exchange visualised. Argo/UML [12] does provide a small amount of additional architecture-oriented notation, notably C²-style communication "buses", but this is inadequate for large system design.

Some tools and notations have been developed specifically for software architecture modelling or have had more comprehensive architecture modelling capabilities added. Examples include PARSE-DAT [4], ViTABaL [2], Clockworks [1], SAAMTool [3], and JComposer architectural aspects [8]. These typically support only limited kinds of architectural abstractions. PARSE-DAT focuses on process-oriented views of architectures, ViTABaL on tool-based abstraction and SAAMTool on structural composition. ClockWorks [1] uses component diagrams but with additional architecture "annotations", representing caching, concurrency and replication. Clockworks supports some code generation from these annotations to help automate realisation of such facilities from their visual specifications. PARSE-DAT provides reasonably high level views of processes and inter-process communication [4] but lacks support for OOD or for code generation, and is limited to basic process views. Most other architecture modelling approaches also focus on basic process and/or program structure (such as SAAMTool) [3]. Most tools that provide architecture notations lack support for dynamic visualisation of realised systems using equivalent notational representations.

Few CASE or other tools provide architecture model analysis and verification mechanisms or integration and reverse engineering support. PARSE-DAT, ViTABaL and ClockWorks provide some analysis support, but limited to specific kinds of domains. Argo/UML provides design critics but these mostly focus on low-level OOD model evaluation heuristics. Argo's critics cannot currently be extended in any way by users, which is problematic if new architectural modelling features need to be added to the environment. Specialised analysis tools, such as those for CSP [26], allow the validation of (limited) architectural models via formal analysis. Some Architecture Description Language support tools, such as those for Wright [21] and Rapide [27], also focus on formal specification of architectural styles and support reasoning about the characteristics of such styles. However our key interest is not so much in the characteristics of certain architectural styles or approaches, but in supporting developers in modelling and validating the use of such styles/approaches on development projects.

Dynamic visualisation of systems is useful for developers to understand system correctness (i.e. to debug them), and to understand higher-level system behavioural characteristics that can not be easily determined from static architecture design views and analyses. Various tools support object visualisation and object structure querying, but lack higher level abstractions [18, 3]. Others support higher-level visualisation over object graphs, generating call graphs, map visualisations and 3D visualisations [28, 19], but these focus at only the object level, and are hard to scale and interpret for large, distributed applications. Various program visualisation systems have been developed, many offering high-level animations and visualisations of algorithms and object structures. These include VisualLinda [29], Rose/Architect [6], The Software Bookshelf [30], and PvaniM [31], and those using 3D call graphs and object trees [19, 18]. While these visualisations are useful, they typically bear no relation to static architecture modelling languages and views, and are thus difficult to formulate and interpret. ViTABaL [2] provides dynamic views of reasonably high-level system components ("toolies") and their relationships but developers must construct these views only from running components, limiting its usefulness.

4 Overview of SoftArch

The above deficiencies in current CASE and related approaches to software architecture design motivated us to develop the SoftArch environment. SoftArch provides an extensible visual notation for software architecture modelling support and an environment that allows models to be constructed and refined. Analysis agents guide, advise and/or constrain architects, and templates

allow reuse of a variety of software architecture refinements. A visualisation facility reuses architecture modelling views to provide high-level visualisation of the dynamics of running systems. Fig. 3 outlines these basic SoftArch capabilities.

Architects build up software architecture designs drawing on a set of extensible meta-model architecture element types (components, connectors and annotations) (1). These element types describe possible kinds of architectural components, connectors and annotations, and the properties of these elements constrain the use of such entities. For example, for E-commerce systems like the travel planner, thin-clients, web servers, http requests, databases and their inter-connections are common modelling elements a designer draws upon to model important parts of their particular problem domain.

SoftArch supports the notion of refinement of software architecture elements and groups of elements into successively more detailed, numerous and lower-level element groupings (2). Properties of high-level architectural components constrain the kinds of refinements and properties at lower levels of detail. For example, a high-level travel system component such as “Customer Clients” might be refined to “Map Visualiser”, “Itinerary Editor”, and “Desktop Applications”. A conceptual group of system functionality such as “Itinerary Management” might be refined to “Itinerary Clients”, “Itinerary Servers”, “Database Server”, “Web server” with associated connectors and annotations.

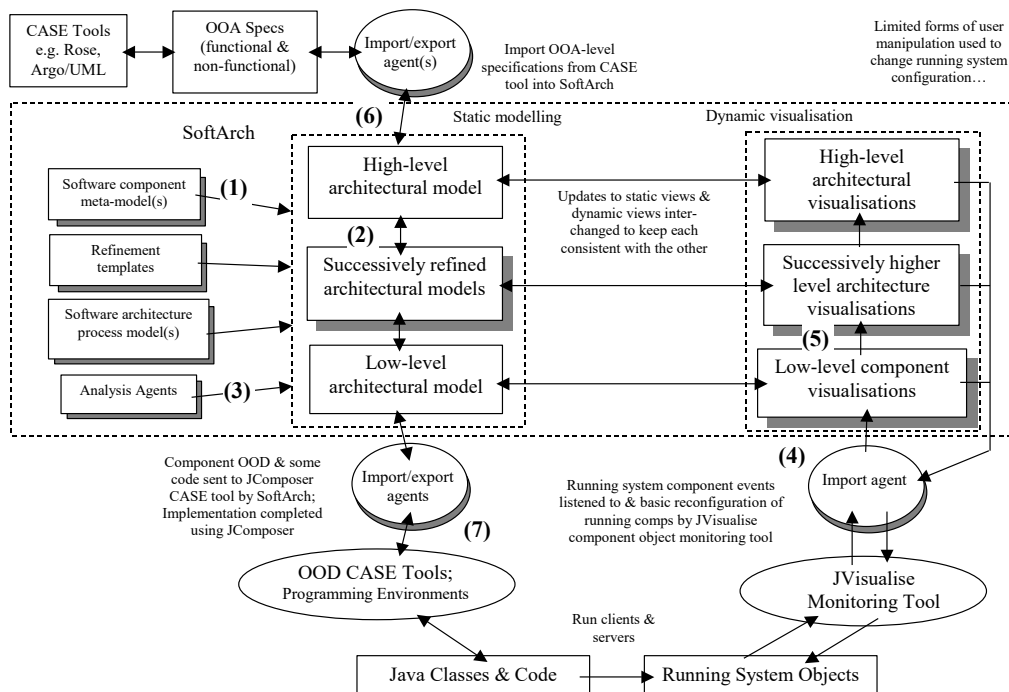


Fig. 3. Overview of SoftArch architecture design modelling, analysis and dynamic visualisation approach.

Analysis agents monitor architecture model changes and advise architects on model correctness i.e. if various meta-model specified constraints between element types and property values are being adhered to (3). These give the architect feedback (in various ways) as they model and refine a system. This feedback is typically unobtrusive, though architects can request immediate notification of constraint violation or can manually request agents run model checking.

Understanding the behaviour of the architecture of a system like the travel planner when it is running is challenging. To help architects validate the run-time properties of their architectures, we capture low-level object events (method calls, property changes, component events) from running systems that are forwarded to SoftArch (4). In SoftArch, OOD-level architecture components are located based on event annotations and information about the running system is passed to their abstractions (i.e. higher-level components). Static SoftArch visualisation views are copied and annotated to convey this running system information to developers e.g. to highlight created/not created processes, indicate number/size/timing of messages between components etc (5).

We deliberately designed SoftArch not to be a complete CASE tool, but rather to share information with CASE tools and programming environments. Import/export tools support linkage

between SoftArch and OOA, design and implementation tools. OOA models allow software architects to capture functional and non-functional requirements in SoftArch and ensure software architecture models meet these, or at least are annotated with this information (6). For example, travel planner functional requirements might be imported from Rational Rose™ OOA descriptions. Partial OOD models and some code fragments (implementing socket protocols, database access, ORB API calls etc.) are exported from bottom-level architecture components e.g. to OOD CASE tools or programming environments, like JBuilder™, to implement the travel planner system (7). Reverse engineering of OOD models into SoftArch allows developers to abstract higher-level architectural models from their code.

5 Static Architecture Modelling

In this and the following sections we describe and illustrate the static architecture modelling, architecture analysis and dynamic architecture visualisation capabilities of SoftArch. Consider again an architect wanting to design a software architecture for the travel planning system from Section 2. The architect needs some (ideally extensible) architectural abstractions to work with and a visual modelling notation to represent these abstractions in multiple views of the architecture. These views provide perspectives on the architecture at varying levels of detail.

5.1. Modelling Notations and Meta-Model

SoftArch uses the concepts of architecture *components*, *associations* (connections) between components, and *annotations* on components and associations. The types of component architects might use include processes, data stores, data management processes (e.g. database servers), machines and devices, and OOA and OOD-level objects and classes. Associations include data usage associations, event notification/subscription, message passing, and process synchronisation links. Annotations include data used, events passed, messages exchanged, protocol used, caching, replication and concurrency information, process control information, ports and so on. Each of these architectural elements can have associated properties. These could include information on services, security approaches, data size, transaction processing speed, data, message and event exchange details, and so on. Property values can be simple numbers, enumerated values, strings or value range constraints.

Architectures are made up of complex, inter-connected *elements* (i.e. components, associations and annotations). Visualising these inter-connected parts provides architects with key viewpoints on their architecture's design. To give this perspective, we provide architects with a visual architecture modelling language to represent architecture elements. Fig. 4 (a) shows some of the basic notational elements in our architecture modelling visual language. This notation has been developed over several years to represent various architectural abstractions in both our teaching and research projects [2, 8]. We chose this visual language for architecture modelling to enable developers to capture a wide range of features, to be relatively simple yet expressive, to be relatively easy to extend as needed, and to be able to tailor the appearance of visual elements to their needs. A wide variety of notational symbol and display characteristics can be changed by architects, such as iconic appearance, size, colour, shading etc. A UML-style representation of architecture using deployment and component diagram-like icons is also supported [11, 12], though we have found that these lack sufficient expressive power and diversity for most architecture design.

The nature of software architecture design means architects often want to incorporate new modelling abstractions (new types of components, associations and annotations) into their models for different problem domains, or to better capture important abstractions. A way of doing this is to allow architects to extend the modelling abstractions available (and visual notations used to represent these) within SoftArch. We use a software architecture meta-model to describe all of the types of components, associations, annotations and properties of these different elements available for use by an architect. To enable architects to easily extend this meta-model SoftArch provides a simple visual language to describe the meta-model, illustrated in Fig. 4 (b). Ovals represent architecture component types, horizontal bars inter-component association types, and labelled vertical arrows annotation types. Dashed, arrowed lines between types indicate refinement e.g. a process can be refined into a client or server process. Solid arrowed lines indicate association relationships e.g. a data manager may have data usage relationships with any architecture element.

For example, when developing the travel planning system, an architect may find a useful architecture abstraction is missing e.g. web server, servlet, desktop application, http protocol etc. In order to use this abstraction during modelling the architect may choose to add this element type to the meta-model, refining it from an existing, more generic element, and linking it to other elements. Properties and constraints can be specified for the element type to enable detailed description and reasoning about its correct usage by analysis tools.

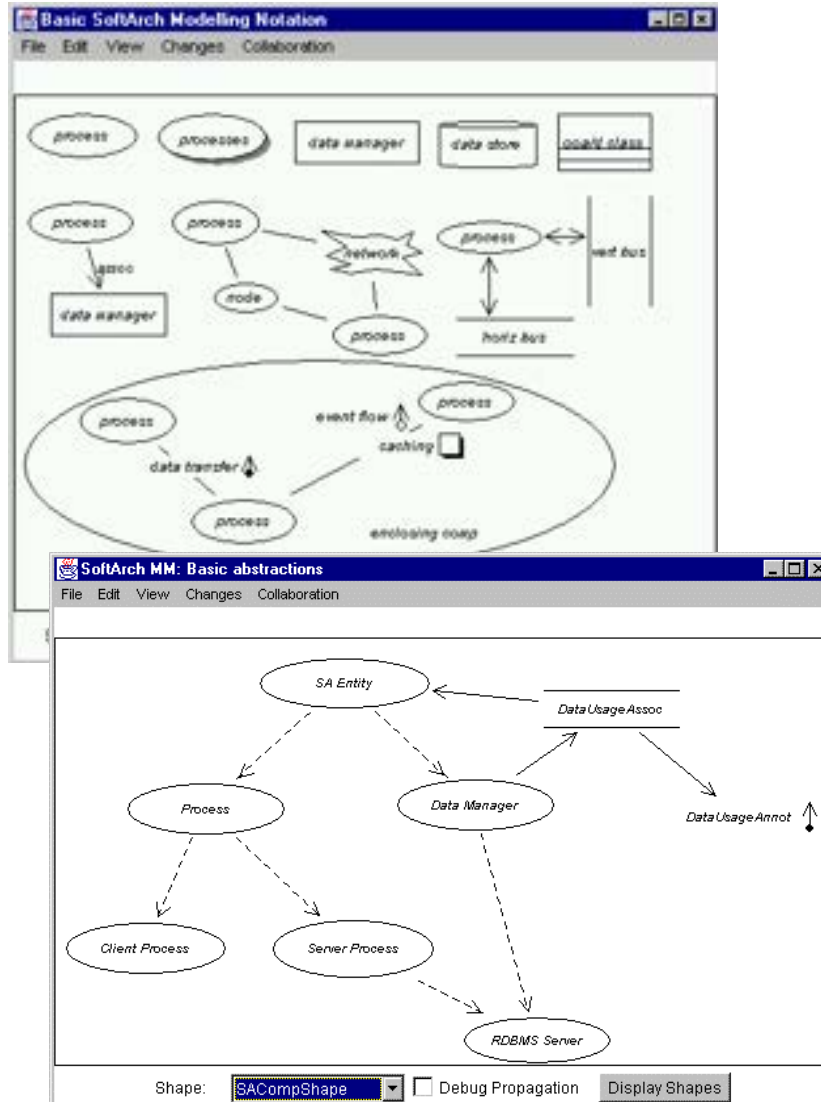


Fig. 4. (a) Some examples of our SoftArch visual modelling notation, and (b) part of a SoftArch meta-model.

5.2. Architecture Modelling Example

To illustrate the use of the SoftArch notation and environment for architectural modelling, consider the modelling of the travel planning application described in Section 2.

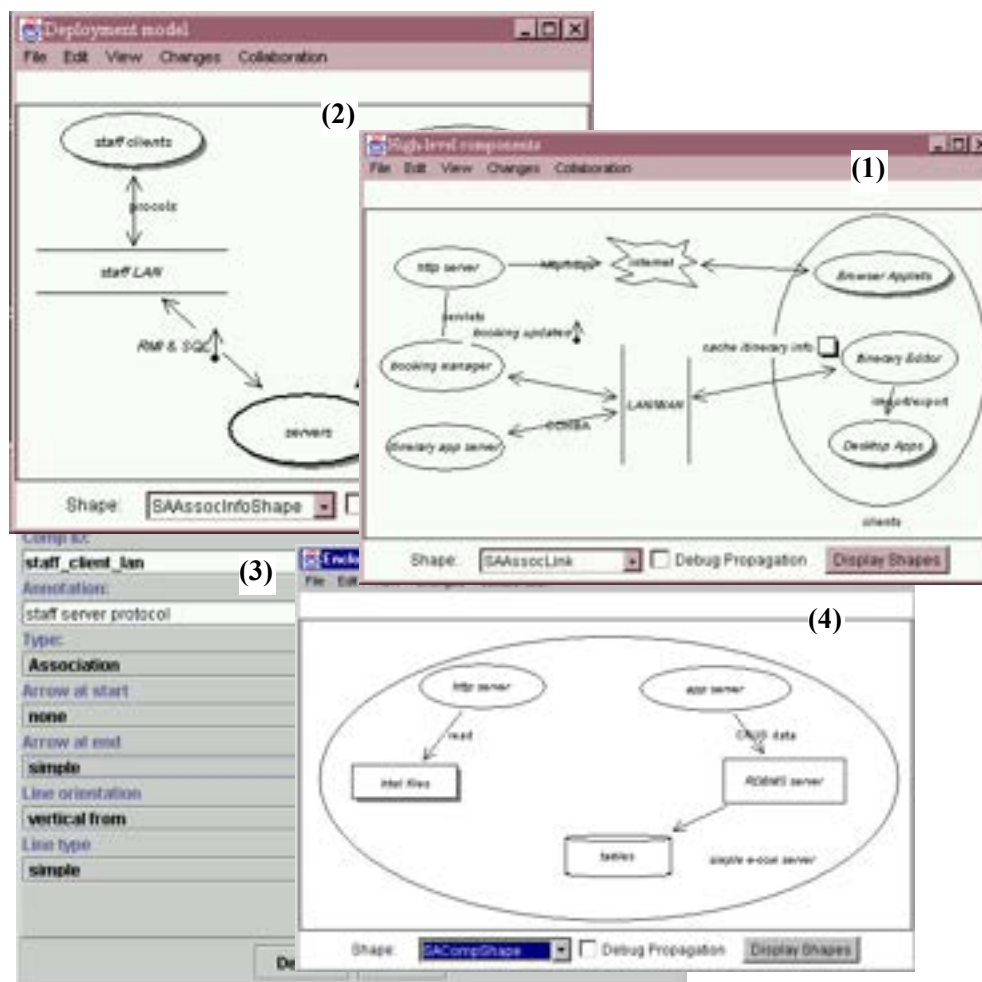


Fig. 5. Examples of architecture modelling in SoftArch.

To begin with, an architect initially imports an OOA functional and non-functional specification from a CASE tool into SoftArch or defines this information directly into SoftArch itself (using simple OOA-level class and function element types). The architect then sketches out a high-level architectural model for the planned system, ensuring the general characteristics of this model meets the OOA specification. For example, the travel planner has to support a number of concurrent users, customers require a web interface, travel planner components need to communicate with both client-side desktop applications and server-side enterprise applications, and various data processing, network and host machine characteristics need to be adhered to by the architecture (performance, reliability, cost and so on).

Fig. 5 (1) and (2) show two such high-level views of the travel itinerary planning system architecture. In (1) the architect has represented the parts of the system as three groups of “processes” – “staff clients”, “customer clients” and “servers”. They have indicated the staff client applications are connected to the servers via a LAN association, the customer clients via a WAN (i.e. internet) association. Annotations add further information such as the expected protocols for communicating with the servers. In (2), the architect has represented the major server-side and client-side processing components making up the system and high-level associations and annotations between these. Such views allow software architects and system designers to describe the basic architectural approach of the system using simple architectural elements. Some elements may appear in more than one view, and some views may show both structural characteristics and dynamic event/message/data exchange. In Fig. 5 (3) the architect is viewing/setting properties associated with an association between staff clients and the enterprise servers, which may include visual appearance and non-functional properties of the element. These architecture diagrams are not always built from scratch. Reusable template views, such as that shown in Fig. 5 (4) provide a means for them to reuse best-practice or common architectural structures. In this example the architect considers reusing a simple server-side “e-commerce” system organisation, made up of http, application and RDBMS servers and associated data. Architects can copy any view for reuse

as a template and may select an appropriate template and have SoftArch copy this into their project, automating linking of abstract elements to new refined elements copied from the template. Change management between templates and copied views is supported [32].

5.3. Architecture Refinement

Once an architect has designed high-level architectural views capturing the essence of their system architecture, they usually wish to flesh this high-level architecture out in more detail, ultimately down to partial OOD-level class and relationship abstractions. Such refinement allows a system to be visualised from multiple perspectives, some showing basic architectural elements, others detailed views of parts of a system, with traceability supported between high-level and low-level abstractions. There are three ways to refine an architectural model in SoftArch: enclosing components within another (all enclosed elements are refinements of the enclosure), adding sub-views for an element (all elements in the view are refinements of the view owning element), and specifying explicit refinement links between elements.

For example, the architect may decide to further specify what “staff clients” are required, and so create a sub-view for the “staff clients” component in Fig. 5 (1). Fig. 6 (1) shows this sub-view. All components in this view, except for “staff lan”, are refinements of the higher-level architecture component (“staff clients”) which owns the sub-view. This allows architects to “hide” this level of detail and drill down to it by double-clicking the “staff clients” icon to show its refinements. A component may have several sub-views, with refined components shown in more than one sub-view. In this example, “staff clients” is refined to “staff booking client”, “staff itinerary editor” and “staff desktop apps” processes. Annotations indicate a CORBA protocol supports booking client to server communication, the itinerary client caches itinerary data and itinerary update events are “pushed” to the itinerary editor. The “staff lan” component is shown in this refinement for context (what the “staff clients” sub-components are connected to) but the “<...>” component name annotation indicates it’s a linkage component and not a refinement component in this sub-view.

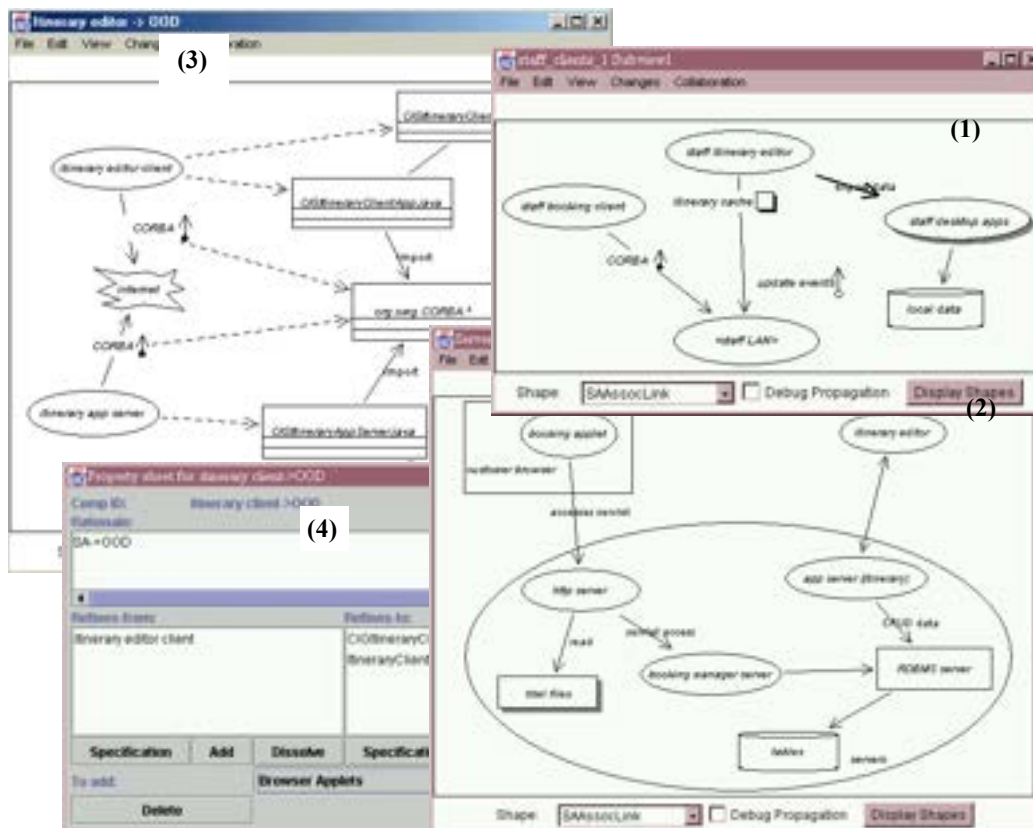


Fig. 6. Example architecture refinements in SoftArch.

The software architect has made two further refinements in this example. In diagram (2), the “servers” component from Fig. 5 (1) has been refined by using it to enclose more specific server-side components, relationships and annotations. All enclosed components, associations and

annotations are refinements of the “servers” component. The architect has chosen to use the enclosing of other components so the context of the refinement is shown on the diagram with the refined components. Note that this refinement has reused the template architecture from Fig. 5 (4), copying the template and the architect renaming and adding to it for use in this modelling application. In diagram (3), several architecture components describing the itinerary management part of the system, on the left hand side, have been refined to OOD-level class components (using UML class diagram notation) on the right hand side. This was done by the architect adding explicit refinement links (the dotted arrows). The dialogue shows refinement information for the `itinerary_client` to OOD classes refinement. OOA-level classes and services can also be described in SoftArch (usually imported from a CASE tool rather than defined in SoftArch itself) and refinement links from OOA classes to architectural components can be made.

SoftArch provides a number of additional modelling facilities our architect may choose to use. These include dynamic behaviour representation, showing components, connectors and dynamic information annotations including data, message, control and event flow/relationships and timing. We have also incorporated some PARSE-DAT and ViTABaL [4, 2] dynamic component assembly constructs, allowing architects to model architectures whose components and connectors evolve at run-time. Our software architect can also model the provided and required services of components and connectors, allowing us to check (from static design models, at least) that these are met. Dynamic properties of components can be modelled and compared against actual performance and other run-time collected measurements.

6 Architecture Analysis

Supporting modelling of software architectures and refinements is not sufficient to enable software architects to produce quality, consistent architecture models for complex systems. Software architecture analysis tools are also needed, including support for checking such things as: all components are linked to others, all components are suitably refined from OOA-level specifications to OOD-level class realisations, sensible and consistent associations and annotations have been used, valid property values have been set, and provided and required services between linked components are consistent.

For example, our architect may have specified a variety of views of their architecture as outlined in the previous section. However, a number of problems may be present in these architecture designs:

- Some architecture elements may not be associated with others or may be associated in invalid ways.
- Some elements may not have all required property values specified for their types.
- Some architecture elements may not be refined from higher-level components or refined to lower-level components. This indicates inconsistency between refinement levels.
- Some elements may require specific kinds of services/properties from related elements, but these are not provided.
- Some architecture elements may be invalidly refined from or to others.
- Some elements may provide services or properties that are not used by related components and should be.

Our meta-model architecture modelling types specify various types of validity information (valid associations, properties and property value ranges, refinement relationships and so on). In addition to checking such architecture element type usage correctness, the architect may want to be provided with feedback on the use of various architecture elements or parts of models in various situations i.e. usage guidelines.

In order to assist architects in static validation of their architecture models SoftArch provides an extensible set of model analysis agents. These monitor changes to an architecture model and give feedback in various ways to the software architect - immediate report of error; unobtrusively adding error notes to an “error list”; or producing a report when the architect requests one. Fig. 7 shows basic approaches our model checking agents use to detect and report on errors. Some agents e.g. Agent #1 simply detect a change to a model element (or changes to elements related to that model element) against meta-model type information. For example, an agent may check if the component has all required property values set, is refined from another element, or has required associations to/from other kinds of elements (and these are valid). The agent reports any

discrepancies between the meta-model element type specifications and the model element instances it can find. Agents can subscribe to changes from single architecture model elements or groups of related elements. Agents can filter out changes they are not interested in e.g. an association checking agent only detects “establish or dissolve relationship” events. We use a change event dispatcher to detect model changes based on element type, forwarding these to subscribing model checking agents.

Other model checking agents, such as Agent #2, may detect changes to one or more components and then compare their inter-connectivity and property values against the agent’s “correctness” template, reporting any error(s). These templates are simply SoftArch architecture modelling templates, like Fig. 5 (4), with a property for all components, associations, refinement links, annotations saying if they are optional/required, and additional element property constraints. The checking agent validates the changed model elements by comparing them against the template. For example, a “valid E-commerce architecture” checking agent may check for the presence of a web server, application server, database and suitable connections, possibly with suitably constrained element properties.

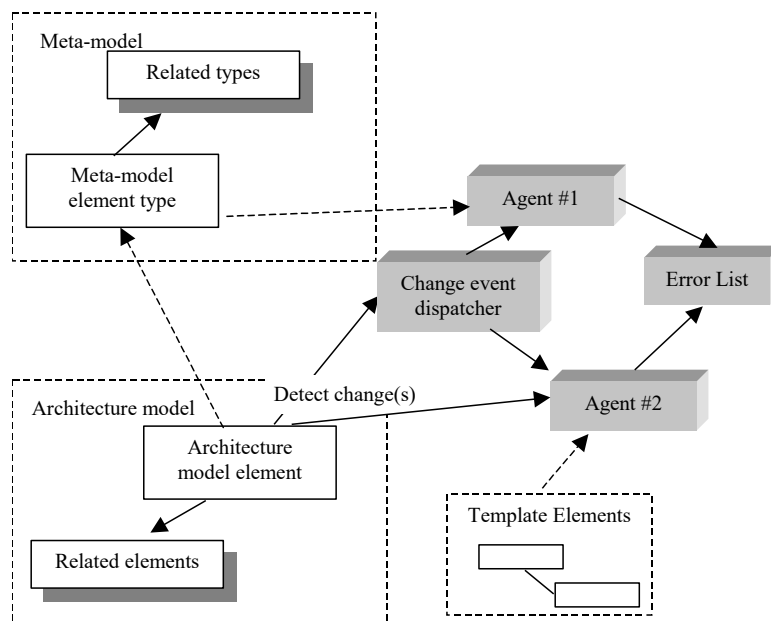


Fig. 7. Analysis agent control, reporting and visual specification.

Fig. 8 illustrates how our software architect can control agent behaviour and view agent error message reports via an agent control panel (shown on the left). In this example, the architect has all of the agents configured as “critics” i.e. agents watch model changes and add messages to a critic message dialogue (shown at the bottom). Several example messages are shown, indicating various discrepancies between meta-model type specifications and actual usage of architecture elements in the model. The architect can ignore these and continue modelling, select one of these and correct it, or correct a number of these errors and leave others. Inconsistent architecture models can be modified with inconsistencies tracked in this way.

7 Dynamic Architecture Visualisation

Once an architecture model design is complete, the software architect typically exports partially specified OOD-level components to a CASE tool and/or programming environment. Developers complete the implementation using these tools. Software architects may then want to analyse the actual behaviour of their implemented architecture, or to analyse a reverse-engineered architecture, using SoftArch-style abstractions. For example, after completing travel system implementation our architect may want to study the behaviour of their architecture in practice, possibly to identify and correct problems, possibly to record performance and other characteristics for use when developing other architectures in the future. In order to support dynamic architecture analysis we have developed support for monitoring and visualising performance information within SoftArch. The approach we use is to capture running system events (such as method calls, object creation, time to complete method call etc), and forward these events to SoftArch, tagging

them with information about the lowest-level SoftArch model element they are associated with (OOD class/method, low-level architecture abstraction etc). We then aggregate these events within SoftArch i.e. associate events with SoftArch elements and then “pass them up” refinement hierarchies, summing them at each level in the hierarchy. This gives a multi-level analysis of overall running system performance measures. We present this aggregated performance information to the architect using annotated design diagrams (though architects can also develop diagrams specifically for aggregating performance information in different ways).

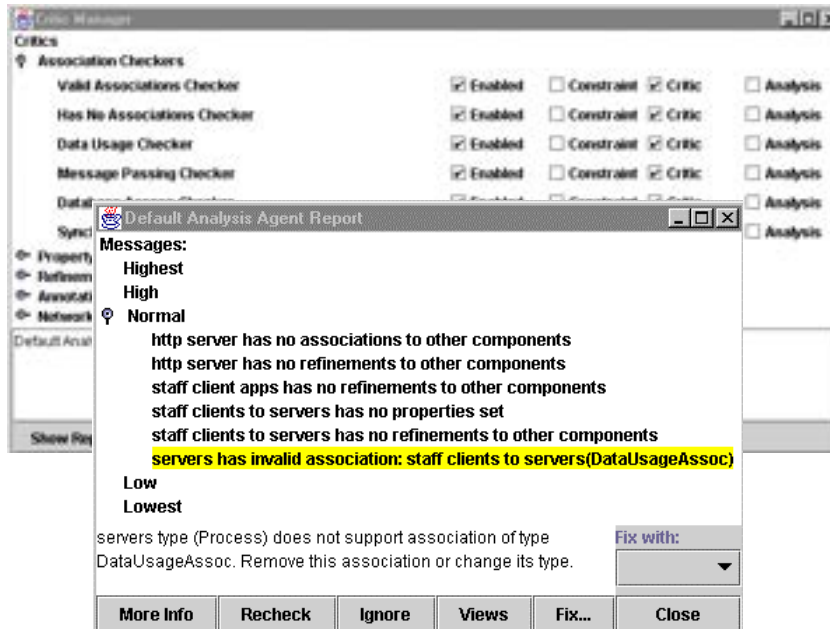


Fig. 8. Example of architecture design critics.

For example, after starting up the itinerary editor servers and one itinerary client staff application, Fig. 9 (1) shows a dynamic visualisation using a top-level architectural view in SoftArch. This visualisation represents the number of components created so far. The servers component is dark (five server-side objects created), staff clients is lightly shaded (one staff application is running) and customer applets very lightly coloured (no objects of types that are refined from this high-level component running on non-staff hosts have been created). This kind of visualisation is useful for software architects to determine what processes in an architecture have so far been created, and to determine relative densities of objects etc in their realised architecture. Views can be animated to show density increasing as a system runs. As with other visualisations, scaling is used to show relative object and process creation densities, data transfer totals and number of events exchanged, and so on. The architect can change properties of the visualisation (colour, scaling, elements updated, frequency of update etc) as they require.

Fig. 9 (2) shows the server-side view of the itinerary planning system, annotated to illustrate relative method calling and event propagation densities. This helps our architect identify bottlenecks and performance problems. Association line thickness for indicates method/event propagation density across the relationship (however implemented). Border thickness of component icons indicates number of methods/events in, while background colour indicates internal method/event calling. This visualisation shows the architect that the booking applet and booking manager server are moderately used and perform a limited number of internal calls. The Itinerary editor and its application server perform many more internal calls, and in this scenario generate more client<->server interactions. The RDBMS is moderately heavily used (caching in the application servers reduces its load), while the http server's servlet is moderately used and performs few internal calls.

Sometimes the software architect wants point-value information about specific architecture element performance measures, as shown in Fig. 10 (1). This shows data for the application server objects at a snap shot in time. Summarised information, in this example a bar graph of number of method calls to the “itinerary app server” component, an object making up the application server process, has been generated by exporting data to MS Excel™, shown in Fig. 10 (2). Developers can request that inter-component communication tracing information be recorded for selected

architectural elements, and can review these. Fig. 10 (3) shows an example of such information presented in a dialogue. Each method invocation of the application server has been recorded, and the developer can examine this trace.

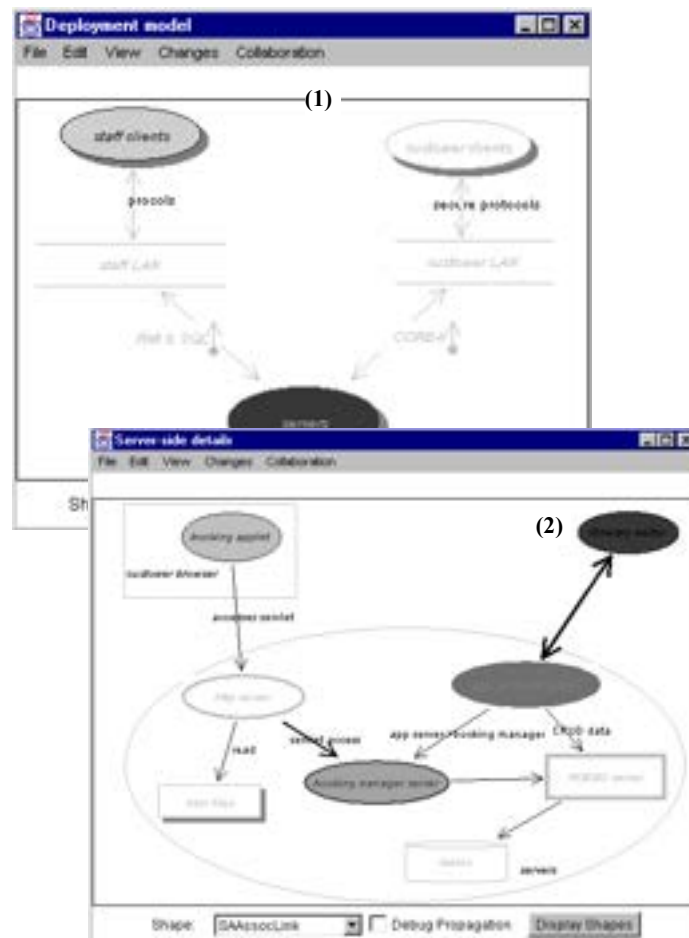


Fig. 9. Dynamic visualisation of running system in SoftArch.

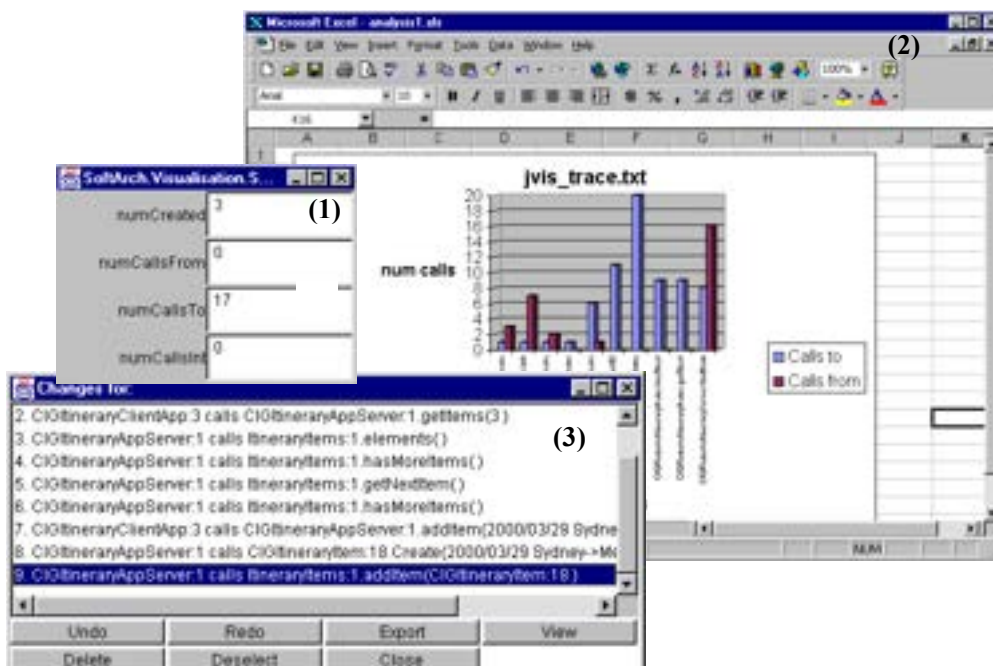


Fig. 10. Detailed and summarised performance information.

8 Design and Implementation

Figure 11 illustrates the basic architecture of SoftArch which comprises:

- The SoftArch modelling and meta-modelling tools. These provide a set of meta-model architecture element type abstractions, reusable model templates and software architecture model views.
- Serendipity-II workflow system [32]. This provides process models and project plans for guiding use of SoftArch, and supports visual plug-and-play of analysis agent parts for use by SoftArch.
- JComposer component-based CASE tool [9]. JComposer provides abstractions for modelling the specifications and designs of software components, and we use JComposer to complete design and implementation of SoftArch architecture models. JComposer also allows reverse-engineering of architecture designs from existing component code (currently Java files).
- Import/export tools for communicating with 3rd party CASE tools. To date, we have built XMI-based tools for communication with Argo/UML and a comma-separated value data exchange tool with MS Excel™.
- The JVisualise dynamic component debugging tool [9]. JVisualise allows us to monitor running system events and aggregate these into SoftArch. It also allows us to perform limited manipulation of running component-based systems.
- Communication/event handling by the JViews software bus [23].

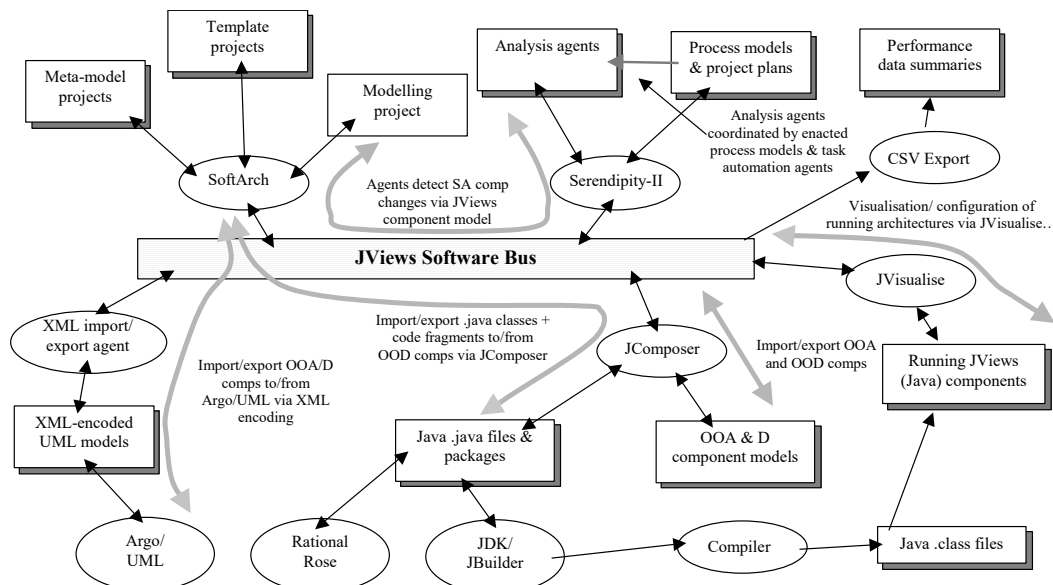


Figure 11. Architecture of SoftArch and related tools.

SoftArch's meta-model, model and visual editing views are implemented using our JComposer tool's meta-CASE framework, which generates classes that specialise our JViews component-based architecture for multi-view, multi-user environment construction [9]. SoftArch is thus a component-based system and able to be integrated with other component-based tools by JViews component facilities. SoftArch provides multiple views of software architecture models with a centralised repository and view consistency mechanism. It provides a variety of collaborative work facilities, including synchronous and asynchronous editing of views, version merging and configuration management.

SoftArch maintains a set of meta-model projects that define the allowable components, associations, annotations and property types for a model. A set of reusable refinement templates (which are copiable SoftArch model views) allow reuse of common architectural refinements. A modelling project holds the model of the software architecture currently under development, including all views, architectural elements and projects. This information is represented as JViews software components using a three-level architecture, as illustrated in Figure 12. The bottom layer is a canonical representation of data; the middle view information shown in each view; and the top

a set of user interface components forming an editor and icons. The extensible meta-model can be visually extended allowing new modelling abstractions and constraints to be dynamically added and reused. The templates can be copied and instantiated into a model for reuse, with changes able to be propagated between model and source template. The graphical user interface icons and connectors used by SoftArch are user-tailorable, allowing architects to change and extend the appearance of their modelling views. This is particularly important if architects add new meta-model abstractions - they can also extend their appearance of the SoftArch visual icons to distinguish new modelling element types from others.

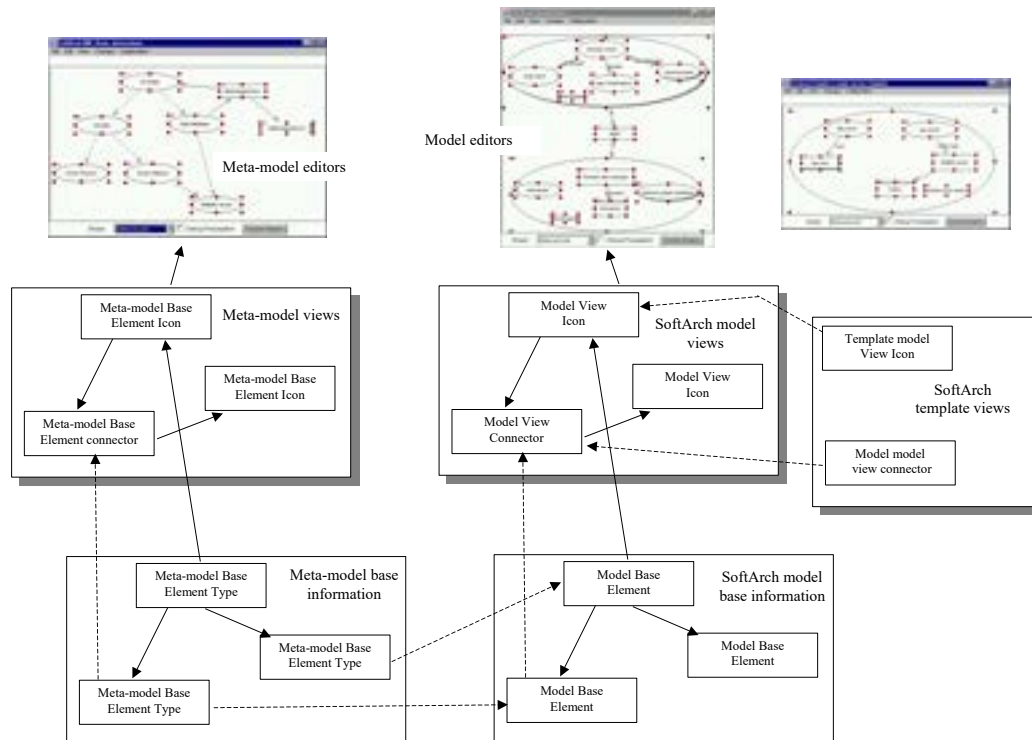


Figure 12. 3-level architecture of SoftArch.

Many tools exist which provide object-oriented analysis and design capabilities. Our own JComposer is one such example, but others include CASE tools like Rational Rose [11] and Argo/UML [12]. We originally planned SoftArch as an extension to JComposer, but decided it would be more useful as a stand-alone tool, that could ultimately be used in conjunction with other, 3rd party CASE tools. SoftArch requires constraints from an OOA model, particularly non-functional constraints like performance parameters, robustness requirements, data integrity and security needs and so on. These constrain the software architecture model properties that needs to be developed in order to realise the specification. These also influence the particular architecture-related design decisions and trade-offs software architects need to make. Similarly, a SoftArch architectural model is little use on its own, but needs to be exported to a CASE tool and/or programming environment for further refinement and implementation. Some code generation can potentially be done directly from a SoftArch model description e.g. some middleware and data management code. When reverse engineering an application, an OOD model will need to be imported into SoftArch and a higher-level system architecture model derived from it. Ultimately an OOA specification may be exported from SoftArch to a CASE tool. Thus SoftArch must support OOA and OOD model exchange with other tools, and ideally some code generation support.

JComposer and SoftArch interact to achieve OOA import and OOD design export and code fragment generation for SoftArch. Generated .java class source code files can be used in tools like JDK and JBuilder, and changes reverse engineered back into JComposer and then into SoftArch. We initially used a JComposer component model as the source for SoftArch OOA-level specification information. JComposer allows not only functional requirements to be captured, but has the additional benefit of requirements and design-level component “aspects”, which are used to capture various non-functional requirements. We developed a component that supports importation of basic component and aspect information into SoftArch from a JComposer model,

using JViews' inter-component communication facilities to link SoftArch and JComposer. However, rather than add OOD and code generation support to SoftArch itself, we leveraged existing support for these in JComposer. SoftArch uses JComposer's component API to create OOD-level components (classes) in JComposer, and instructs JComposer to generate code for these to produce .java files. We have prototyped a data interchange mechanism to enable SoftArch to exchange OOA and OOD models with Argo/UML using an XML-based encoding of UML models.

We use our JVisualise component monitoring tool to request running components send it messages when they generate events. SoftArch instructs JVisualise to send it these low-level component monitoring events, which are mapped onto SoftArch OOD-level architecture elements using JComposer-generated Java class names. SoftArch allows users to view information about running components using higher-level SoftArch views, as OOD-level components in SoftArch must have refinement relationships to higher-level architecture elements in these views. We extended JComposer-generated OOD models and code to include additional monitoring components to intercept data and communication messages and to annotate these with the source SoftArch elements to which low-level generated component events are related. JVisualise uses these to provide its event and message monitoring and control support.

9 Experience

We have used SoftArch to model a number of small and medium-sized system architectures. These have included the travel planner discussed here, a business-to-customer on-line retailing system, an on-line video store library system, and an on-line micro-payment system. We have also used it to help reverse-engineer the architectures of several component-based, distributed systems. We have used SoftArch as the architecture modelling tool component of SoftArch/MTE, a performance test-bed generator which takes SoftArch models and generates performance test-bed code [33]. This includes the generation of JSP, ASP, CORBA, Enterprise JavaBean and .NET implementations of SoftArch-modelled systems, with stubs generated for clients, servers and database access code to allow architecture and middleware performance analysis to be automatically carried out.

We have carried out two usability evaluations of SoftArch to assess its support for architecture development. These involved a combination of graduate students, researchers and industry practitioners modelling system architectures, and in the second scenario generating performance test-bed code for analysis. These evaluations have demonstrated SoftArch provides a number of useful facilities not found in comparable CASE or development environments. These include the ability of designers to refine architecture models in various ways that supports much richer architectural representation and reasoning and traces architectural design decisions clearly from OOA to OOD. Analysis agents that provide incremental feedback to architects while tolerating varying amounts of inconsistency during architecture design allow for more flexible architecture development while providing continuous validation guidance. Visualisation of running systems using high-level abstract views provides much easier to understand performance information and more rapid feedback than most other approaches. When coupled with our performance test-bed generator component, the SoftArch architecture visualisation support can also be reused to provide high-level visualisation of automatically-generated performance analysis tests. Users of SoftArch liked its flexible design notation but commonly suggested using "more UML-style notations". They also found the import/export of architecture models between different tools e.g. Argo/UML, SoftArch and JBuilder, to be cumbersome. The current visual language used to define architecture design critics was found to be too difficult for most users of SoftArch surveyed.

We are working on characterising a wider range of architectural components for various domains, such as for embedded systems, real-time systems and E-commerce systems, and defining meta-models, notational symbols and reusable templates to enable easier modelling of such systems. We are building further analysis agents to make better use of architecture component performance measurements, giving developers improved estimates of likely run-time behaviour of architecture models. We plan to incorporate more structured architectural properties, characterised using our aspect-oriented engineering method [8], allowing developers to characterise their architectural component and association characteristics using systemic aspects and for SoftArch to perform consistency analysis of inter-aspect relationships. The JComposer-based code generation and monitoring is quite effective at providing developers with high (and low) level dynamic architectural component performance and trace information. However, it requires our JViews-

based component architecture be used to realise running distributed applications. We are currently working on a code generator that uses XML-encoded UML models to generate parts of systems and 3rd party profiling tools whose traces will be acquired and aggregated by SoftArch to provide run-time performance information. We plan to use this code generation and profile aggregation to allow developers to rapidly develop middleware test bed systems to validate architectural model performance properties via rapid architecture prototyping and performance analysis.

10 Summary

Modelling, validating and visualising complex system architectures is a challenging development activity. SoftArch provides a set of tools enabling architects to model rich knowledge about their architectures using an extensible set of architectural abstractions and visual notations. The extensible SoftArch meta-model allows developers to define new, specialised architectural components for particular domains, while the tailorable visual notations allow developers to represent their architectures in a wide variety of ways. Architectural component refinement allows developers to refine their architectures from analysis-level specifications to design-level implementation descriptions in a controlled and traceable fashion. Architecture analysis in SoftArch uses basic consistency checks and comparison to best-practice element usage templates to help inform developers of the quality of their models. This provides support for proactive architectural refinement during modelling. Dynamic visualisation of running systems is supported by aggregating captured trace events and displaying this information by annotating static modelling view components. This approach presents developers with run-time architecture performance metrics in a context they can readily interpret.

Acknowledgements

Support for this research from the University of Auckland Research Committee and the New Zealand Public Good Science Fund is gratefully acknowledged. The helpful comments of the anonymous referees on earlier drafts of this paper are also appreciated.

References

1. T.C.N. Graham, C.A. Morton, and T. Urnes, ClockWorks: Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages and Computing*, (July 1996), 175-196.
2. J.C. Grundy, J.G. Hosking, ViTABaL: A Visual Language Supporting Design by Tool Abstraction, In Proc. of the 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany, September 1995, IEEE CS Press, pp. 53-60.
3. R. Kazman, Tool support for architecture analysis and design, In Proc. of the Second Int. Workshop on Software Architectures, ACM Press, 94-97.
4. A. Liu, Dynamic Distributed Software Architecture Design with PARSE-DAT, In Proc. of the 1998 Australasian Workshop on Software Architectures, Melbourne, Australia, Nov 24, Monash University Press.
5. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.
6. A. Egyed and P. Kruchten, Rose/Architect: a tool to visualize architecture, In Proc. of the 32nd Hawaii Int. Conf. on System Sciences, January 1999, IEEE CS Press.
7. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.
8. J.C. Grundy, Supporting aspect-oriented component-based systems engineering, In Proc. of 11th Int. Conf. on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 388-395.
9. J.C. Grundy, W.B. Mugridge, J.G. Hosking, Static and dynamic visualisation of component-based software architectures, In Proc. of 10th Int. Conf. on Software Engineering and Knowledge Engineering, San Francisco, June 18-20, 1998, KSI Press.
10. T. Urnes and T.C.N. Graham, Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations. In Proc. of Design, Specification and Verification of Interactive Systems (DSV-IS'99), 1999.
11. T. Quatrani, *Visual Modeling With Rational Rose and Uml*, Addison-Wesley, 1998.
12. J. Robbins, D.M. Hilbert, and D.F. Redmiles, Extending design environments to software architecture design, *Automated Software Engineering* 5 (July 1998), 261-390.
13. N. Stankovic and K. Zhang, K. Towards Visual Development of Message-Passing Programs, In Proc. of 1997 IEEE Symposium on Visual Languages, IEEE CS Press.

14. B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi, Visual Patterns + Multi-Focus Fisheye View: An Automatic Scalable Visualization Technique of Data-Flow Visual Program Execution. In 1998 IEEE Symposium on Visual Languages, Halifax, Canada, September 1998, IEEE.
15. J.C. Grundy and J.G. Hosking, Directions in modelling large-scale software architectures, In Proc. of the 2nd Australasian Workshop on Software Architectures, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
16. J. Leo, OO Enterprise Architecture approach using UML, In Proc. of the 2nd Australasian Workshop on Software Architectures, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
17. M. Beaumont, and D. Jackson, Visualising Complex Control Flow. In 1998 IEEE Symposium on Visual Languages, Halifax, Canada, September 1998, IEEE.
18. T. Hill, J. Noble, Visualizing Implicit Structure in Java Object Graphs, In Proc. of SoftVis'99, Sydney, Australia, Dec 5-6 1999.
19. S.P. Reiss, A framework for abstract 3-D visualization, In Proc. of the 1993 IEEE Symposium on Visual Languages, IEEE CS Press.
20. R.J. Walker, G.C. Murphy, J. Steinbok, and M.P. Robillard, Efficient Mapping of Software System Traces to Architectural Views, In Proc. of CASCON'2000.
21. R. Allen and D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology, July 1997.
22. M. Shaw and D. Garlan, Software Architecture : Perspectives on an Emerging Discipline, Prentice Hall, 1996.
23. J.C. Grundy, W.B. Mugridge, J.G. Hosking and M.D. Apperley, Tool Integration, Collaboration and User Interaction Issues in Component-based Software Architectures, In Proc. of TOOLS Pacific'98, Melbourne, Australia, Nov 28-30 1998, IEEE CS Press.
24. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, P. Pattern Oriented Software Architecture : A System of Patterns, Wiley, 1996.
25. B. Wagner, I. Sluijmers, Eichelberg, D. and P. Ackerman, Black-box Reuse within Frameworks Based on Visual Programming, In Proceedings of the. 1st Component Users Conf., Munich, July 1996, SIGS Books, pp. 57-66.
26. A. Parashkevov and J. Yantchev, ARC - A Tool for Efficient Refinement and Equivalence Checking for CSP, In Proc. of the 1996 IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing, Singapore, June 11-13, 1996.
27. D.C. Luckham, L.M. Augustin, J.J. Kenney, J. Veera, D. Bryan, and W. Mann, Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering **21**(April 1995), 336-355.
28. J.G. Hosking, Visualisation of object-oriented program execution, In Proc. of 1996 IEEE Symposium on Visual Languages, IEEE CS Press.
29. H. Koike, T. Takada, and T. Masui, VisuaLinda: A Framework for Visualizing Parallel Linda Programs, In Proc. of the 1997 IEEE Symposium on Visual Languages, IEEE CS Press.
30. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, M. and K. Wong, The Software Bookshelf, IBM Systems Journal 36 (November 1997), 564-593.
31. B. Topol, J. Stasko and V. Sunderam, PVaniM: A Tool for Visualization in Network Computing Environments, Concurrency: Practice & Experience **10** (1998), 1197-1222.
32. J.C. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, IEEE Internet Computing 2 (September/October 1998), IEEE CS Press.
33. J.C. Grundy, Y. Cai, Y. and A. Liu, Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In Proc. of the 2001 Int. Conf. on Automated Software Engineering, San Diego, Nov 25-28 2001, IEEE CS Press.

3.5 A Visual Language for Design Pattern Modelling and Instantiation

Maplesden, D., Hosking, J.G. and **Grundy, J.C.**, A Visual Language for Design Pattern Modelling and Instantiation, *Chapter 2 in Design Patterns Formalization Techniques*, Toufik Taibi (Ed), Idea Group Inc., Hershey, USA, March 2007, pp. 20-43. DOI: [10.4018/978-1-59904-219-0.ch002](https://doi.org/10.4018/978-1-59904-219-0.ch002)

Abstract: In this chapter we describe the Design pattern modeling language, a notation supporting the specification of Design pattern solutions and their instantiation into UML design models. DPML uses a simple set of visual abstractions and readily lends itself to tool support. DPML Design pattern solution specifications are used to construct visual, formal specifications of Design patterns. DPML instantiation diagrams are used to link a Design pattern solution specification to instances of a UML model, indicating the roles played by different UML elements in the generic Design pattern solution. A prototype tool is described, together with an evaluation of the language and tool.

My contribution: Co-designed approach, wrote some of the software the approach based on, co-supervised Masters student, co-authored significant parts of the paper, co-lead investigator for funding for this project from FRST

A Visual Language for Design Pattern Modelling and Instantiation

David Maplesden
Orion Systems Ltd
Mt Eden, Auckland, New Zealand
Phone: +64 9 638 0600
Fax: +64 9 638 0699
Email: David.Maplesden@orion.co.nz

John Hosking
Department of Computer Science
University of Auckland,
Private Bag 92019 Auckland, New Zealand
Phone: +64 9 3737599
Fax: +64 9 3737453
Email: john@cs.auckland.ac.nz

John Grundy
Department of Computer Science and Department of Electrical and Computer Engineering
University of Auckland,
Private Bag 92019 Auckland, New Zealand
Phone: +64 9 3737599
Fax: +64 9 3737453
Email: john-g@cs.auckland.ac.nz

A Visual Language for Design Pattern Modelling and Instantiation

Abstract

In this chapter we describe the Design Pattern Modelling Language, a notation supporting the specification of design pattern solutions and their instantiation into UML design models. DPML uses a simple set of visual abstractions and readily lends itself to tool support. DPML design pattern solution specifications are used to construct visual, formal specifications of design patterns. DPML instantiation diagrams are used to link a design pattern solution specification to instances of a UML model, indicating the roles played by different UML elements in the generic design pattern solution. A prototype tool is described, together with an evaluation of the language and tool.

Keywords: *CASE tools, Language Constructs, Modeling Languages, Special Purpose Languages, Class diagram, Meta Model, Object-Oriented Design*

INTRODUCTION

Design patterns are a method of encapsulating the knowledge of experienced software designers in a human readable and understandable form. They provide an effective means for describing key aspects of a successful solution to a design problem and the benefits and tradeoffs related to using that solution. Using design patterns helps produce good design, which helps produce good software (Gamma et al, 1994).

Design patterns to date have mostly been described using a combination of natural language and UML-style diagrams or complex mathematical or logic based formalisms which the average programmer find difficult to understand. This leads to complications in incorporating design patterns effectively into the design of new software. To encourage the use of design patterns we have been developing tool support for incorporating design patterns into program design. We describe the DPML (Design Pattern Modelling Language), a visual language for modelling design pattern solutions and their instantiations in object oriented designs of software systems. We have developed two prototype tools, DPTool and MaramaDPTool, realising DPML and integrating it within the Eclipse environment. Significant contributions of this work include the introduction of *dimensions* as a proxy for collections of like design pattern participants and the instantiation of patterns into designs rather than directly into code. These both fit naturally with model driven design approaches.

We begin by describing previous work in design pattern tool support. We then overview DPML and describe its use in modelling design pattern solutions and pattern instantiation. We then discuss two prototype tools we have developed to support the use of DPML, together with an evaluation of their usability. We then discuss in more detail the rationale and implications of the design choices we have made in designing DPML and the potential for more general applicability of some of those design features before summarising our contributions.

PREVIOUS WORK

Design patterns, which describe a common design solution to a programming problem, were popularised by the seminal “Gang of Four” book (Gamma *et al*, 1994) and Coplien’s *Software Patterns*

(Coplien, 1996). Design patterns have become very widely used in object-oriented software development, and their influence has spread to areas of software development other than design, such as the development of analysis patterns (patterns in the analysis phase) and idioms (language specific programming patterns). Design patterns are typically described using a combination of natural language, UML diagrams and program code (Gamma et al, 1994; Grand, 1998). However, such descriptions lack design pattern-specific visual formalisms, leading to pattern descriptions that are hard to understand and hard to incorporate into tool support. The UML standard for modelling design patterns relies upon UML profiles and the UML meta-model (Object Management Group, 2006). This presents difficulties for modelling design patterns, particularly because they are constructed using similar concepts to object models and hence are simply prototypical examples of that object model. This does not allow enough freedom to model patterns effectively. Design pattern representations look like existing UML models and linking pattern elements to standard UML elements is often not supported. Several attempts have been made to improve design pattern representation with UML (e.g. Mak et al, 2004; Fontoura et al, 2002; Guennec et al, 2000) but all use conventional UML diagram representations or minimal extensions. Stereotypes and related approaches to delineating patterns makes the diagrams considerably more complex and discerning pattern elements from standard UML elements is difficult.

Lauder and Kent (1998) propose an extension to UML to aid in “precise visual specification of design patterns”. They use a 3-layer model with a visual notation for expressing models. The notation is an amalgam of UML and “Constraint Diagrams” a notation to visually specify constraints between object models elements. A second notation expresses object dynamic behaviour and can represent generalised behaviour of design patterns. We found their notation difficult; the differentiation between the diagrams at different levels was unclear and it seemed difficult to understand the reason why some abstractions were made at one level and not another.

There have been a number of approaches proposed for alternative visual representations for design patterns. LePUS (Eden, 2002) uses a textual higher order monadic logic to express solutions proposed by design patterns. Primitive variables represent the classes and functions in the design pattern, and predicates over these variables describe characteristics or relationships between the elements. A visual notation for LePUS formulae consists of icons that represent variables or sets of variables and annotated directed arcs representing the predicates. LePUS’ basis in mathematics and formal logic makes it difficult for average software developers to work with and provides a weak basis for integrated tool support, being well removed from typically used design and programming constructs. LePUS tool support is based on Prolog and lacks support for the visual mean that while diagrams are compact, they are difficult to interpret with a high abstraction gradient. LePUS concentrates solely on defining design pattern structures, and has no mechanism for integrating instances of design patterns into program designs or code. Mak et al (2003) have proposed an extension to LePUS which addresses pattern composition.

Florijn et al (1997) represent patterns as groups of interacting “fragments”, representing design elements of a particular type (eg class, method, pattern). Each fragment has attributes (e.g. classname), and roles that reference other fragments representing pattern relationships, e.g. a class fragment has method roles referencing the method fragments for methods of that class. The fragments actually represent instances of patterns. Pattern definitions are represented by prototype fragment structures; a one-level approach to defining patterns where the patterns, kept in a separate repository, are identical to

the pattern instances in the fragment model. This approach lacks support for the definition of design patterns and also a strong visual syntax. The single level architecture means patterns are only defined as prototypical pattern instances. We argue that concepts exist at the pattern level that do not at the pattern instance level, thus patterns can't be specified in the most general way using only prototypical instances, i.e. you cannot specify all patterns in the most general way using only prototypical instances. Their approach to pattern definition also has no formal basis. It is limited to defining patterns only relative to Smalltalk programs represented in the fragment model. We feel it would be advantageous to define an exact unambiguous meaning for the pattern representation in use so that it can be discussed without confusion and applied appropriately to a range of programming languages.

RBML (Kim et al, 2003; France et al, 2004) adopts a similar to approach to ours, and has been influenced by initial work we have presented in this area (Mapelsden et al, 2002). It uses a meta-modelling approach to the specification of pattern representation, extending the UML metamodel to achieve this. They place more emphasis on behavioural specification than we have in the development of DPML, which has focussed more on structural representations of patterns. They also adopt a cardinality approach to specification of multiplicities as opposed to our dimension concept. The tradeoffs involved are discussed further in Section 0.

Some approaches use textual rather than visual languages (e.g. Reiss 2000; Sefika et al, 1996; Taibi and Ngo, 2003). While these present useful concepts, our interest is in a visual language for modelling design patterns. Domain-specific visual languages like DPML offer a higher level of abstraction and representation, particularly for design-level constructs. We are also particularly interested in applying to design pattern modelling the approach UML (Object Management Group, 2006) takes to object modelling, i.e. providing a common formalism that is accessible to the average designer/programmer, while abstracting away from lower levels of design.

OVERVIEW OF DPML

DPML defines a metamodel and a notation for specifying design pattern solutions and solution instances within object models. The metamodel defines a logical structure of objects, which can be used to create models of design pattern solutions and design pattern solution instances, while the notation describes the diagrammatic notations used to represent the models visually. It is important to stress that DPML can only be used to model the generalised *solutions* proposed by design patterns, not complete design patterns. A complete design pattern also contains additional information such as when the solution should be applied and the consequences of using the pattern.

DPML can be used as a stand-alone modelling language for design pattern solutions or, more commonly, in conjunction with UML to model solution instances within UML design models i.e. whereabouts in the UML model the pattern is used and the various bindings that result from that usage. DPML supports incorporation of patterns into a UML model at design-time, rather than instantiation directly into program code. We feel design-time is the vital stage at which to include design patterns in the software engineering process, the assumption being that if design patterns can be effectively incorporated into the UML object model then converting the object model into code is, relatively speaking, straight-forward.

There were three primary goals for the development of the DPML. Firstly to provide an extension to the UML so that design patterns could be raised to first class objects within the modelling process. Secondly to provide for design patterns some of the same benefits that the UML provides for object oriented modelling: a common language to facilitate the exchanging of models; a formal basis for the understanding of such models; and a basis to facilitate the provision of automated tool support for modelling. By raising design patterns to first class objects in the design process the DPML allows design patterns to become an integral part of the design process. Thirdly, we have aimed for a formalism to express design patterns that is *accessible* to typical programmers. Our aim is for sufficient formalism to provide a robust representation, while avoiding complex mathematical formalisms that restrict the use of our approach to a very small set of mathematically inclined programmers. This is a similar approach to formalism as has been taken in the development of UML. We feel that by providing an easy-to-use yet powerful method for creating and instantiating design pattern solutions, designers will be encouraged to think at higher levels of abstraction about the problems they are facing and come up with more general, re-usable solutions which can, in an iterative manner, be abstracted and then encapsulated in a design pattern for future use.

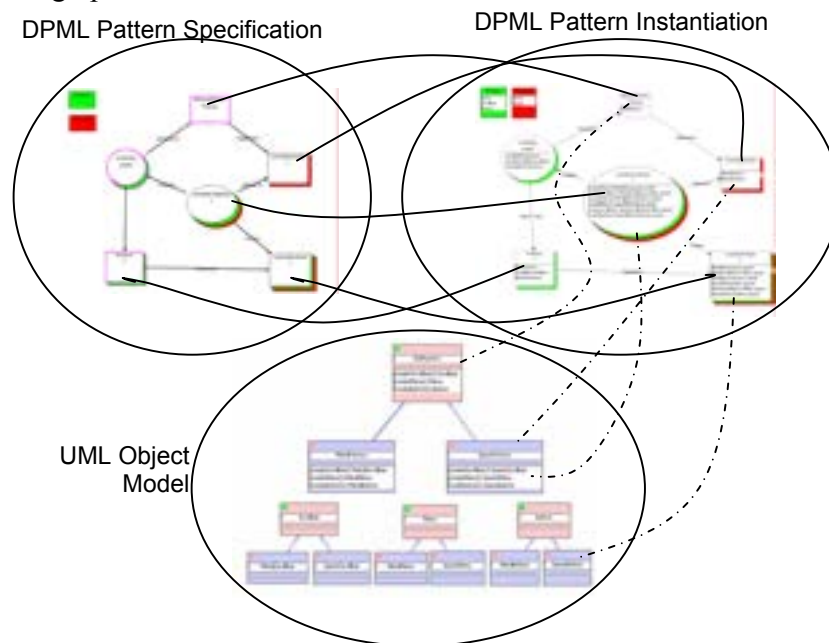


Figure 1: Core concepts of DPML.

The secondary goals of the DPML include firstly, a common standard language for the definition of design patterns that will allow patterns to be exchanged amongst designers in a more readily accessible manner than written text and diagrams, hopefully spreading useful design abstractions and robust designs and improving program design and secondly, providing a basis for tool support for design patterns. DPML has been developed specifically with automated tool support in mind. It is designed to be relatively easy to implement, particularly in conjunction with UML. We have carried out a detailed investigation into the implementation issues for the DPML and the processes that can be supported for working with the DPML and developed two proof-of-concept tool implementations.

The intended uses of the DPML then is to capture areas of good design within an OO model in a design pattern and then reuse them within the same and different models. By specifying the constructs in the design pattern that capture the essential parts of the good design you can ensure that these elements exist every time the design pattern is employed. Indeed by encouraging more experienced designers to create the design patterns to be used by less experienced designers in their work the DPML provides a practical way to encapsulate and reuse the expertise of good designers.

The core concept of DPML (as shown in Figure 1) is that a design pattern specification model is used to describe the generalised design structures of design patterns that are of interest to or useful to the user. This entails modelling the participants (interfaces, methods etc) involved in the pattern and the relationships between them. The user can then use the UML to create an object oriented (OO) model of a system that they are interested in or developing. During the OO modelling process, if the user sees an opportunity to use a design pattern that they have previously defined, they can create an instance of that design pattern from the original definition. The instantiation process consists of linking the roles of the elements in the design pattern with members from the OO model, or creating new model members where required. The well-formedness rules at this stage define which members from the OO model are eligible for fulfilling each role. In this way the user can be sure of creating a valid instance of the design pattern and so be sure of gaining the benefits of using the design pattern.

The design pattern instance model also allows each individual design pattern instance to be tailored. By default a design pattern instance contains members for all objects and constraints on these objects specified by the pattern definition, however certain parts of the pattern may be relaxed or extended on a case by case basis allowing pattern instances that are variations on the base pattern. This recognises the fact that pattern instantiation often involves small adaptations of the pattern to suit the particular context it is being applied to (Chambers et al, 2000).

MODELLING DESIGN PATTERN SOLUTIONS

Pattern Specification

In DPML, design pattern solution models are depicted using Specification Diagrams, the basic notation for which is shown in Figure 2. It should be emphasised that the surface syntax is relatively unimportant (and we have used at least two variants of this in our work). In addition, in our MaramaDPTool the surface appearance can be altered by the user by use of a meta-tool designer. Of much more importance is the abstract metamodel which is described as a UML class diagram in Figure 3. DPML models design pattern solutions as a collection of participants; dimensions associated with the participants and constraints on the participants. A participant represents a structurally significant feature of a design pattern, that when instantiated, will be linked to objects from the object model to realise the pattern. Constraints represent conditions that must be met by the objects filling the roles of the participants in a design pattern instance for it to be considered a valid instance of the design pattern. Dimensions are constructs associated with participants to indicate that the participant potentially has more than one object linked to it in an instantiation. They indicate that a participant represents a set of objects in the object model, instead of just a single object.

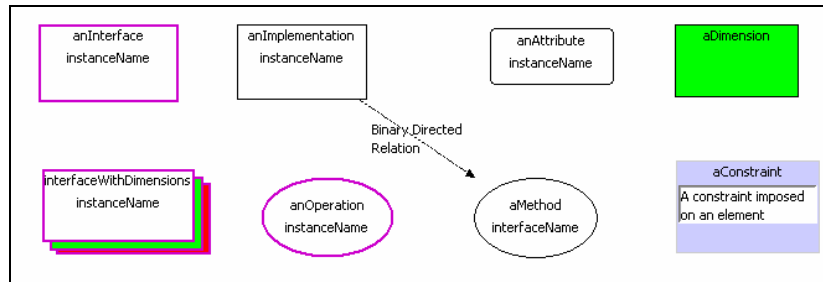


Figure 2: Basic DPML notation

Participants can be interfaces, implementations, methods, operations or attributes. An interface (lighter and thicker bordered rectangle) represents a role that must be played by an object that declares some behaviour i.e. it exhibits an interface or signature in the object model. In a traditional UML class model this means an interface or a class can fill an interface role as both declare a set of operations that provide behaviour. An implementation (darker, thinner bordered rectangle) represents a role played by an object that defines or actually implements some behaviour. In a conventional UML model an implementation would map to a class. The key concept with an implementation is that it defines no interface itself: its type or the declaration of its behaviour is defined entirely by the interfaces it is said to implement. This is different from the traditional concept of a class, which embodies both an interface and an implementation in the one object. This split is designed to allow a clearer definition of roles of the participants in a design. Modellers can specify precisely whether an object is intended to be a declaration of type, an interface, or a definition of behaviour, an implementation. A single object, in the case of a class, can play the roles of both an interface and an implementation.

A relationship similar to the one between interfaces and implementations exists between operations and methods. An operation (lighter thicker bordered oval) is the declaration of some form of behaviour while a method (darker thinner bordered oval) is the definition or implementation of that behaviour. An operation represents a role that must be played by an object in the object model that declares a single piece of behaviour e.g. it can be played by a method or an abstract method in a conventional UML model. A method can only be played by an object that actually defines behaviour and so must be played by a concrete UML method. An attribute (rounded rectangle) is a declaration of a piece of state held by an implementation and defines a role played by a class attribute in the UML model. Constraints are either simple constraints or binary directed relations. Simple constraints (plain text inside a grey box) define a condition specified in either natural language or OCL to be met by the object bound to a single participant. Binary directed relations (lines with arrowheads) define a relationship between two participants, implying a relationship must exist between the objects in the object model playing the roles each of the participants define. The type of the binary directed relation determines the exact relationship that is implied. For example the ‘implements’ relationship between an implementation and an interface implies the object filling the role of the implementation must implement the object filling the role of the interface. Other examples of binary directed relations are *extends*, *realises*, *creates*, *declared in*, *defined in*, *return type* and *refers to*.

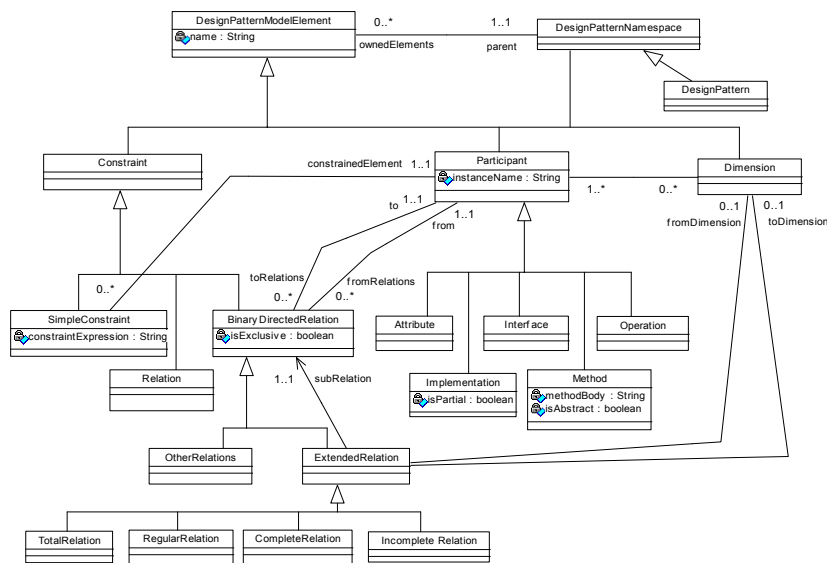


Figure 3: DPML Specification Diagram metamodel

A more complex subclass of binary directed relations is the set of extended relations. These define the mappings of a binary directed relation between participants that have dimensions associated with them. Because these participants have sets of objects associated with them we need to specify how the base relation maps between the sets of objects involved. There are four possible mappings: the relation exists between every possible pair of objects (a *total* relation); exactly once for each object (a *regular* relation); for every object in one set but not necessarily the other (a *complete* relation); and only between one pair of objects (an *incomplete* relation). Extended relations are more fully specified as expressions of the form *extendedRelationName(fromParticipantDimension, toParticipantDimension, subRelationSpecification)*. An example later in this section will illustrate this further.

Dimensions (indicated by a coloured rectangle and coloured shading of participant icons to indicate they are associated with a dimension) specify that a participant can have a set of objects playing a role. The same dimension can be associated with different participants in a pattern and this specifies not only that these participants can have some multiple number of objects associated with them but that this number of objects is the same for both participants.

Pattern Specification Examples

Consider modelling the Abstract Factory design pattern from (Gamma et al, 1995) (Figure 4). This pattern is used by designers when they have a variety of objects (“Products”) which are subclasses of a common root-class to create. A set of “Factory” objects are used to create these related “Product” objects. In this pattern there are six main participating groups of objects. The abstract factory interface declares the set of abstract create operations that the concrete factories will implement. This can be modelled by the DPML with an interface named *AbstractFactory* and an operation named *createOps*. The *createOps* operation represents a set of operations so it has an associated dimension (*Products*) since there is one operation for each abstract product type we want to create. There is also a complete *Declared_In* relation running from *createOps* to *AbstractFactory*. This relation implies that all methods linked to the *createOps* operation in an instantiation of the pattern must be declared in the object that is

linked to the *AbstractFactory* interface. The *Products* interface has the *Products* dimension associated with it to imply there is the same number of abstract product interfaces, as there are abstract *createOps* operations. A regular *Return_Type* relation runs from *createOps* to *Products*, implying each of the *createOps* operations has exactly one of the *Products* as its return type.

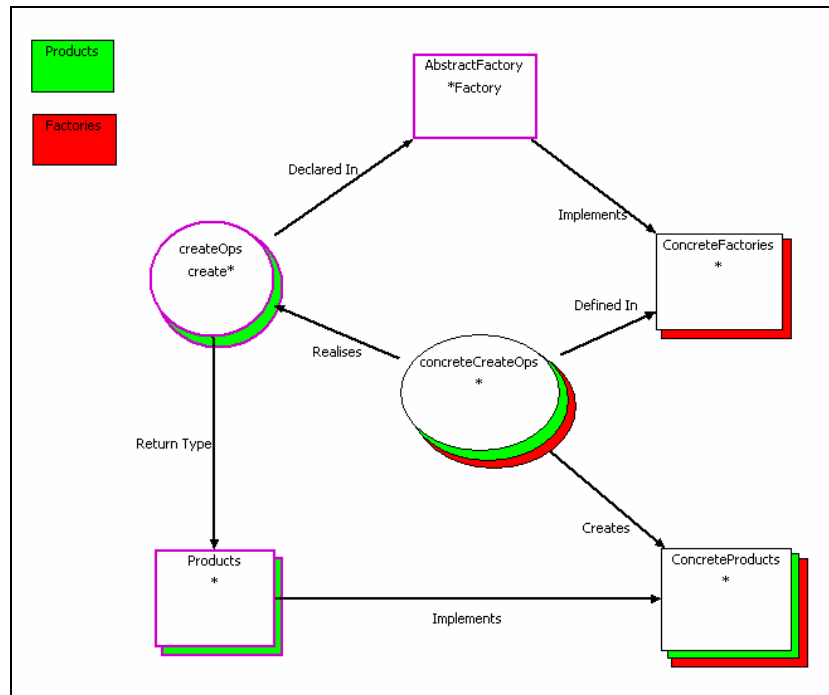


Figure 4: Example specification of Abstract Factory pattern using DPML.

The above set of participants defines the abstract part of the Abstract Factory pattern. The other set of participants define the concrete part of the pattern: the factory implementations, the method implementations that these factories define, and the concrete products that the factories produce. These are modelled by a *concreteFactories* implementation, a *concreteCreateOps* method, and a *concreteProducts* implementation respectively.

The *concreteFactories* implementation has a dimension, *Factories*, to indicate it represents a number of concrete implementations, one for each type of Factory implemented. A complete *Implements* relation runs from *concreteFactories* to *AbstractFactory*, implying all the *concreteFactories* must implement the *AbstractFactory* interface. The *concreteCreateOps* method represents all methods from the set of *ConcreteFactories* that implement one of the sets of *createOps* so it is associated with both the *Factories* and *Products* dimensions. It has a regular relation with a complete sub relation that has a *Defined_In* sub relation running from it to the *concreteFactories* implementation. This extended relation sounds complicated but it implies simply that for every concrete factory there is a set of *concreteCreateOps* that it defines, one member of that set for each *Product*. This can be stated in a more compact expression form, as:

```
regular(Factories, Factories, complete(Products, , Defined_In ))
```

where we see that the regular relation is associated with the *Factories* dimension, ie for each of the *Factories* the complete subrelation holds. This subrelation is between the *Products* dimension on the

concreteCreateOps side and no dimension on the *concreteFactories* side and specifies that every *concreteCreateOp* associated with a *Product* is defined in each *concreteFactory*.

Similarly there is a regular relation with a complete sub relation which has a *Realises* sub relation running from *concreteCreateOps* to *createOps* which implies that for every *createOps* operation there is a set of methods in *concreteCreateOps* that realise it (one in each concrete factory). This can be stated in a more compact expression form, as:

regular(Products,Products , complete(Factories, , Realises).

The overall effect of the two extended relations can be seen in Figure 5 which shows the object structure implied by them.

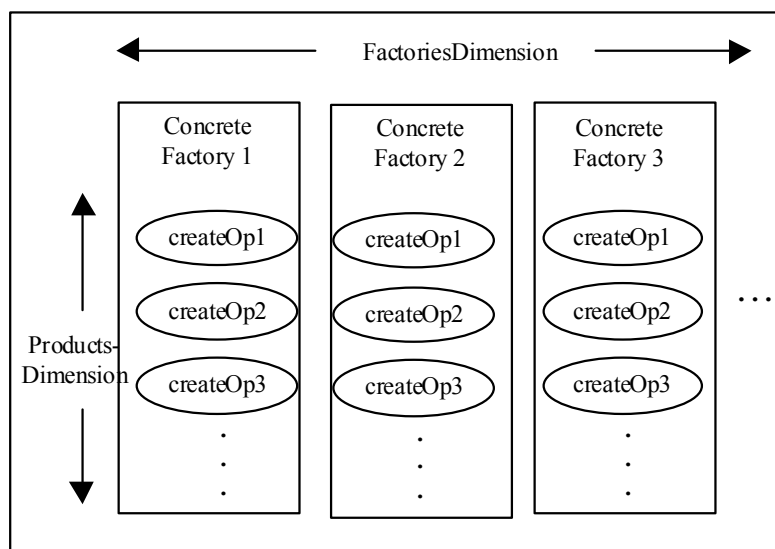


Figure 5: Object structure implied by Defined In and Realised relations associated with ConcreteCreateOps method

Finally the *concreteProducts* implementation has both *productsDimension* and *factoriesDimension* dimensions associated with it. This is because there is exactly one *concreteProduct* for each abstract product and concrete factory i.e. each concrete factory produces one concrete product for each abstract product interface. The *concreteProducts* implementation also takes part in an *Implements* extended relation (regular, complete, Impelments) with the *Products* interface and a *Creates* extended relation (regular, regular, Creates) with the *concreteCreateOps* method. These specify that each *concreteProduct* implements one *Product* interface and that each *concreteProduct* is created (instantiated) in exactly one of the *concreteCreateOps* methods.

As can be seen, the full name of a relation in expression format can be long and can clutter the diagram when the base relation is the important part. So for ExtendedRelations just the name of the base relation can be used on the diagram to improve readability of the diagram, this means the full ExtendedRelation needs to be specified elsewhere (in our proof of concept tools this is specified in a property window). Generally the type of the ExtendedRelation can be deduced from the diagram because relations tend to follow common patterns. Usually (but not always) a relation between two

Participants with the same Dimension will be a RegularRelation and between a Participant with a Dimension and a Participant without a Dimension will be a CompleteRelation. Occasions when TotalRelations and IncompleteRelations are used are much less common and it is advisable in these cases to show the full ExtendedRelation name in the diagram.

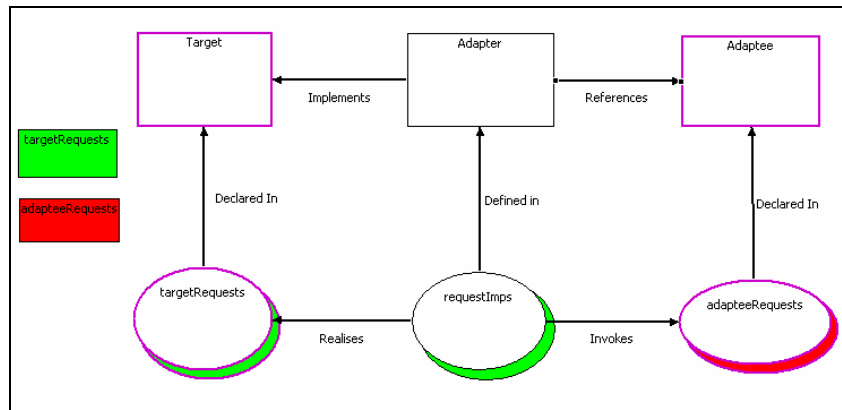


Figure 6: Specification Diagram for the Adapter Pattern

Figure 6 shows another specification diagram, this time for the Adapter Design pattern, also from (Gamma et al, 1995). This specifies how an *Adapter* implementation can be used to map operations specified by a *Target* interface to an *Adaptee*, which has different signatures for its operations. Here we see that the *Target* interface declares a set of *targetRequests*, the set being associated with the *targetRequests* dimension. The *Adapter* implements *Target*'s interface, in the process defining a set of *requestImps* methods, also associated with the *targetRequests* dimension. A regular *Realises* relation between *requestImps* and *targetRequests* specifies that each *targetRequests* operation is realised by one *requestImps* method. The *Adapter* has a *References* relation with the *Adaptee* interface. The *Adaptee* declares a set of *adapteeRequests* one for each member of the *adapteeRequests* dimension. The *Invokes* relation between *requestImps* and *adapteeRequests*, indicates each of the *requestImps* may invoke one or more of the *adapteeRequests*.

Behavioural Specification

In designing DML, we have concentrated on structural specification. However, more dynamic aspects, such as method calling mechanisms, can be represented using an extended form of UML sequence or collaboration diagram. Figure 7 shows an example sequence diagram for the Adapter pattern. This uses standard sequence diagram notation, but includes participants, acting as proxies for the final bound objects. Dimensions, as is the case in the specification diagram, are represented using coloured shading, in this case, the *targetRequests* and *requestImps* invocations are annotated with the *targetRequests* dimension colour and the *adapteeRequests* invocation is annotated with the *adapteeRequests* dimension colour. This component of the formalism needs further development. In particular, it should be possible to use dimensions to indicate looping constructs with a similar set of invocation relationships having a similar set of extended relationship variations as for the structural diagrams. This remains as further work.

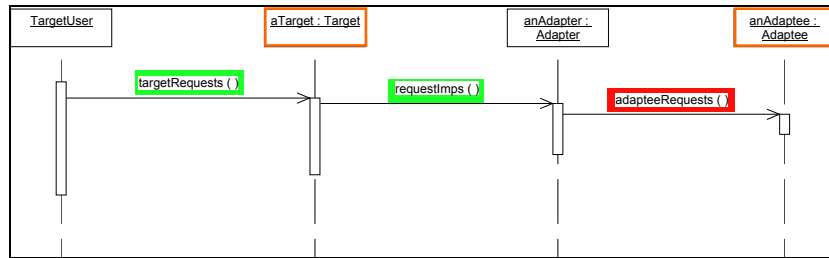


Figure 7: Behavioural Specification for Adapter Pattern

Pattern Instantiation

Instantiation Diagrams provide a mapping from the pattern specification to its realisation in a UML design model. In addition, as mentioned earlier, the instantiation process can adapt a pattern solution through addition of extra participants or modification of existing participants, an important part of our DPML design. Pattern solutions are rarely instantiated directly, the instantiation almost always involves some measure of adaptation or refinement. Accordingly, Instantiation Diagrams have a very similar look and feel to Specification Diagrams. The basic notation is very similar and, in fact, all of the modelling elements (except addition of new dimensions) that are used in Specification Diagrams may also be used in Instantiation Diagrams. These modelling elements are used to model the pattern adaptations. In addition, however, an Instantiation Diagram also includes *proxy* elements (which will typically be the majority of the diagram’s elements). These represent the original participant specifications in an instantiated pattern, but are elaborated with information about the actual UML design elements that they are bound to in this particular instantiation of the pattern. Figure 8 shows the proxy element notation. As can be seen, the syntactic form is similar to that of the specification diagram equivalents. In the case of interface, implementation, operation, method, and relation proxies, they differ from the originals by having dashed borders or lines, and lists of bindings. For constraints, an “inherited” keyword precedes the constraint expression, and for dimensions, a list of the names of the category bindings for that dimension.

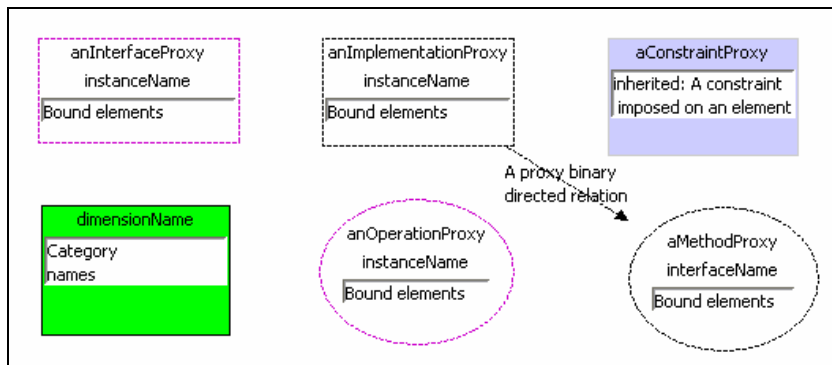


Figure 8: Additional notational elements for Instantiation Diagrams

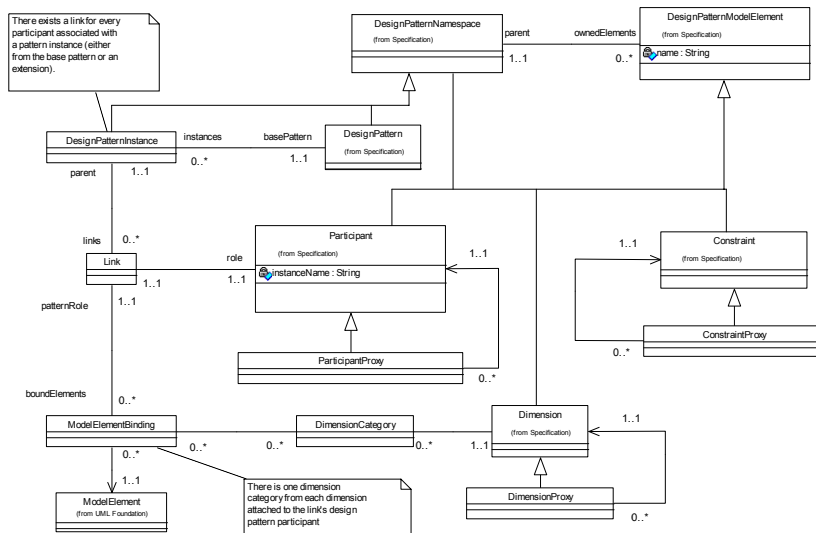


Figure 9: DPML Instantiation Diagram metamodel

Figure 9 shows the metamodel for Instantiation Diagrams. In a DesignPatternInstance (ie the model for an Instantiation Diagram) every Participant (whether they are a ‘proxy’ or ‘real’) has a Link associated with it that maintains a binding from the Participant to some number of UML model elements in an object model (*UMLModelElement* is part of the UML Foundation Package). The names of the bound UMLModelElements are those displayed in the bound elements lists in the proxy participant icons. When model elements are bound to Participants with Dimensions each model element is associated with a DimensionCategory for each Dimension. These DimensionCategories are specified as a simple list of their names which are displayed in the Dimension proxy. The number of DimensionCategories for a Dimension establishes the ‘size’ of the Dimension for the Instance.

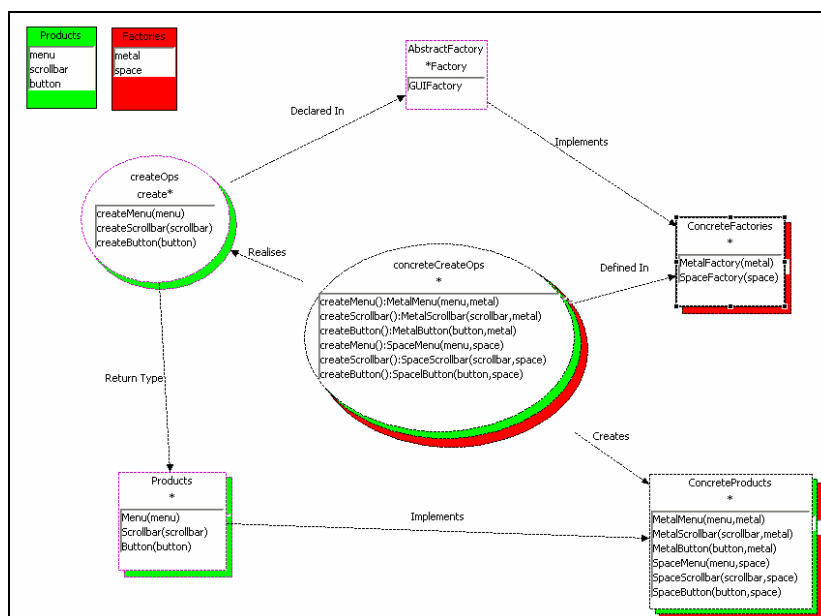


Figure 10: Instantiation Diagram for GUIFactory

Pattern Instantiation Example

As an example, consider the Instantiation of the Abstract Factory pattern shown in Figure 10. Assume we are implementing a GUI toolkit that allows programmers to create a GUI with windows, menus, icons, buttons etc which users can change the look and feel of at runtime and we want to use the Abstract Factory design pattern to do this. The Instantiation Diagram illustrates the bindings for this. The *GUIFactory* UML interface is bound to the AbstractFactory interface participant and this interface is implemented by two *ConcreteFactories*, *MetalFactory* and *SpaceFactory*, one for each dimension category (metal, space) in the Factories dimension (the relevant dimension category name is shown in brackets after the bound element name). *GUIFactory* declares three operations: *createMenu*, *createScrollbar* and *createButton*, one for each of the product dimensions (menu, scrollbar, button) each of which has a return type of the corresponding *Product* type (*Menu*, *Scrollbar*, *Button*). The *concreteCreateOps* method participant has 6 bound elements, a set of 3 methods (each creating a corresponding *ConcreteProduct*) for each of the two *ConcreteFactories*.

Figure 11 shows a UML class diagram representing the UML model elements bound in Figure 10. This could have been independently developed and bindings made manually in the Instantiation Diagram. Alternatively, having specified the dimension category names for each dimension, and some other key bindings (in this case only the *GUIFactory* binding) simple regular expressions (defined in the specification diagram) specifying naming convention patterns combined with the extended relation expressions, and a small amount of manual intervention (notably for incomplete relations) could be used to directly *generate* the bound element names and from them, the bound elements themselves, and thence the equivalent UML model for those elements. For example, the *ConcreteFactories* bindings may be generated by pre-pending the string “Factory” with the Factory dimension category name with its first letter capitalised, while the *ConcreteProducts* bindings may be generated by the cross product of the Factory dimension category names (1st letter capitalised) and the Products dimension category names (1st letter capitalised). The asterisks in the examples indicate names or parts of names which can be generated in this way.

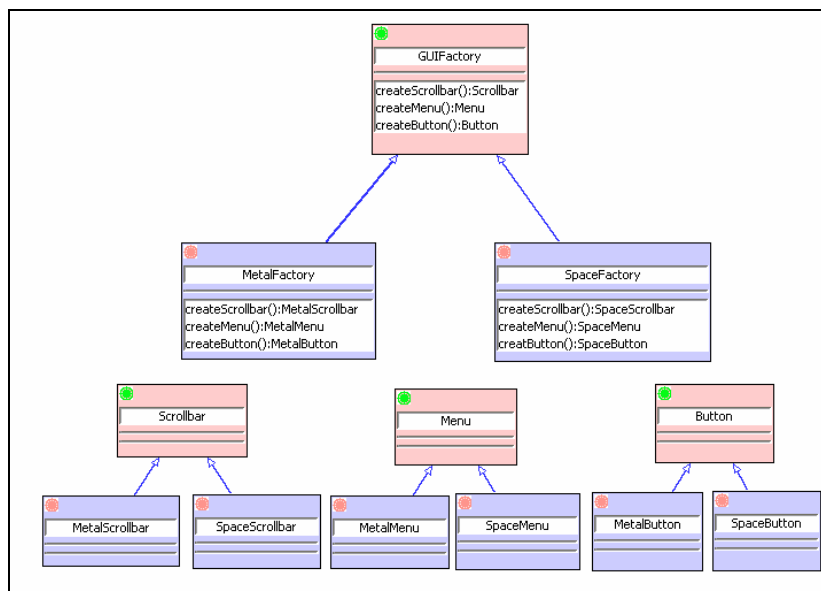


Figure 11: UML Design for GUIFactory

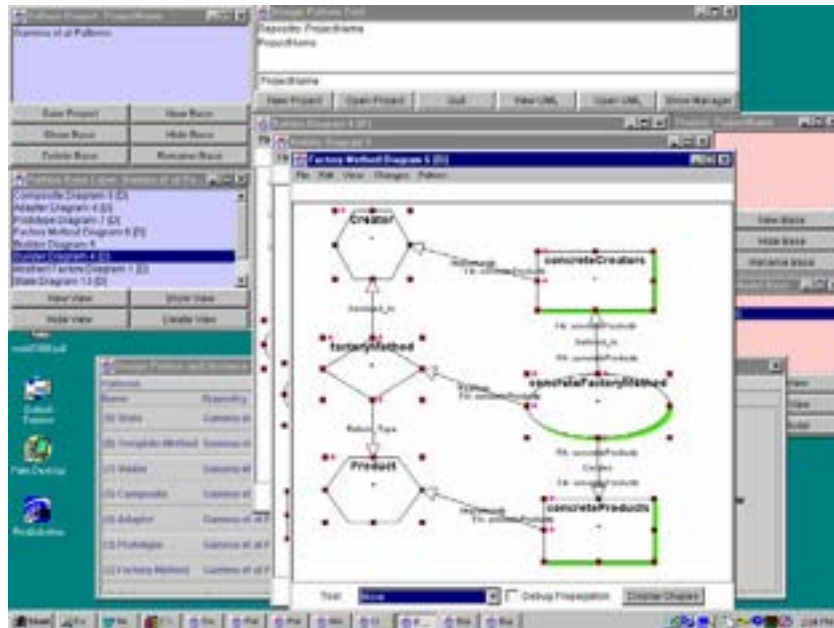


Figure 12: The Prototype DPTool in use

TOOL SUPPORT

We have developed two proof of concept tools to support the use of DPML for pattern modelling and instantiation. The first, reported in (Mapelsden et al, 2002), is a standalone tool, DPTool, implemented using our JViews/JComposer meta-toolset. Figure 12 shows this tool in use. The second, MaramaDPTool, is an Eclipse (Eclipse Foundation, 2006) plugin generated using our Ponamu/Marama meta-toolsets (Zhu et al, 2004; Grundy et al, in press), and which use Eclipse's EMF and GEF frameworks for model and view support respectively. The bulk of the diagrams presented earlier in this paper were generated using MaramaDPTool. Three views of this tool in use are shown in Figure 13. Of the two implementations, DPTool is the most developed in terms of functionality, however MaramaDPTool, based as it is on the Eclipse framework, has far better potential for integration with other programmer productivity toolsets.

Both tools provide the following functionality:

- Modelling views for specification, instantiation, and UML class diagrams, including specification of naming convention patterns.
- For each type of view, multiple views can be modelled, with consistency maintained between the views, meaning that complex patterns or UML models can be broken down into a collection of partial specifications, each contributing to an underlying model
- Support for instantiation of a pattern, through generation of an instantiation diagram with the same layout as the specification diagram it is derived from, but with participants replaced by proxies
- Consistency management between specification and instantiation views, so changes to a specification are reflected in each of the instantiations.
- Support for binding UML model elements to instantiation diagram participants and proxies.

- Consistency management between UML class diagram views and instantiation diagram bindings so that changes to the pattern instantiation can be reflected in the UML class diagrams and vice versa.
- Support for instantiation of multiple, overlapping patterns into a UML model through the use of multiple instantiation diagrams contributing to a common UML model.
- Model management support, to allow saving and loading of models (DPML and UML), undo-redo, etc.

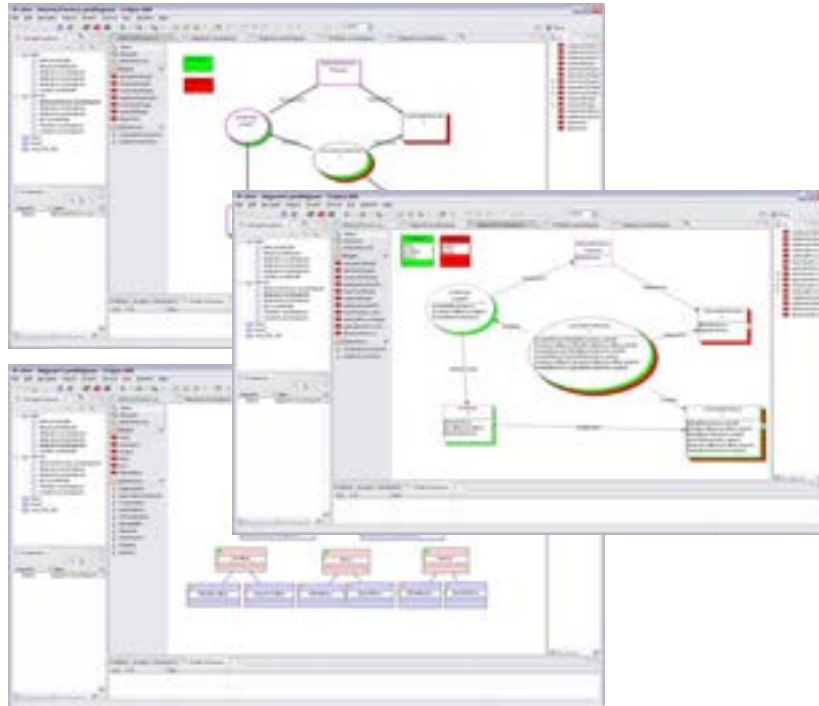


Figure 13: Three views using the MaramaDPTool (top) Design Pattern Specification Diagram (centre) Design Pattern Instantiation Diagram (bottom) UML Class Diagram.

Error Type	Description	ModelElement	Pattern Instance
Model Warning	No view of base model element	MetaFactory.createCar().Car	
Model Warning	No view of base model element	MetaFactory.createScrollBar...	
Pattern Instance Error	The FROM bound element 'MetaFactory.createCar().Car' does not satisfy this relation.	Proxy for Method: createMeth...	TestInstance
Pattern Instance Error	The element "car().Car" has a name which does not match the participant's 'instancename' pattern	Proxy for Operation: createOp	TestInstance

Figure 14: The Error Display in the DPTool

In addition, the JViews based DPTool has better support for pattern realisation and model validation, together with better repository support through provision of a library of predefined patterns. Realisation support includes recommendations on UML model elements that could be validly bound to participants. Validation support checks validity of UML models against the pattern specification, checking for incompleteness, i.e. participants that aren't bound, and inconsistency, i.e. violations of the DPML or UML well formedness rules and violations of the participant naming convention patterns. Errors discovered are displayed in a window, as shown in Figure 14. Validation in the DPTool is a user instigated operation. In our MaramaDPTool, we are implementing an Argo critic style of validation

support which can execute in the background generating a to-do list of identified errors (Robbins et al, 1999).

Pattern Abstraction is the process of identifying a useful or interesting structure or design in an object model and abstracting from that object structure to a suitable DesignPattern model. This mechanism forms the third leg (the first two being pattern instantiation and pattern realisation) of a complete round trip engineering process for design patterns. In Pattern Abstraction a group of elements from an object model are identified and used as a blueprint for creating a DesignPattern model. This is not currently well supported in either of our tools. Currently this requires manual construction of Instantiation and Specification diagrams and binding of participants to the UML model. In our MaramaDPTool we are developing complementary support to the pattern instantiation mechanisms permitting selection of UML model elements and generation of a Pattern Instantiation diagram, which in turn can be automatically abstracted to Pattern Specification diagram. This is more complex than the pattern instantiation process, however. In general it is impossible to create a fully accurate and defined design pattern automatically from a collection of object model elements. This is because there are any number of ways certain arrangements of elements could be abstracted to a design pattern model. A particular difficulty is the recognition of repeating sets of elements as elements that should be represented by Participants with Dimensions. Accordingly, the abstraction support will only be able to automatically perform parts of the abstraction process requiring manual assistance from the user for aspects such as dimension identification. An additional diagram type more explicitly showing binding relationships between DPML and UML elements may be a useful component in solving this problem.

EVALUATION

We have evaluated DPML, and, in particular, the DPTool via an end user based usability study and a cognitive dimensions (Green & Petre, 1996) assessment. Results of these evaluations have been reported in (Maplesden et al, 2002). In brief, the user study, which was qualitative in its nature, showed that the tool was regarded favourably by the survey group, as was much of the language and notation used by the tool. The explicit separation between design patterns, design pattern instances and object models was easy to follow and effective in managing the use of design patterns and users found the tool useful and usable for its primary tasks of creating and instantiating design patterns models. Weaknesses and difficulties noted included a difficulty in understanding the dimension and dimension category concepts due to poor visualisation of these and the lack of annotation capability. Both of these have been addressed in MaramaDPTool, the former by having explicit iconic representation of dimensions and dimension category bindings (these were not in the original notation) and the latter by supporting textual annotations. Pattern abstraction support was also highlighted as a desirable feature, which we are currently addressing in MaramaDPTool. The integration of DPML support into Eclipse via MaramaDPTool allows code generation and consistency management between UML models and target code to be maintained. It also potentially allows us to use MaramaDPTool pattern descriptions with 3rd party UML tools via their XMI-based UML meta-models.

The cognitive dimensions assessment highlighted the strong abstraction gradient and difficult mental operations introduced by the dimension concept. These have been mitigated, as noted above, by more explicit iconic representation of dimensions. However, it is worth noting that DPML is already better in both aspects than other, more formal notations, such as Kent and Lauder's (1998) or LePUS (Eden et al 1998). DPML exhibits good closeness of mapping, and consistency, and DPTool provides good

progressive evaluation support via its validation mechanism. Hidden dependencies are an issue, as is the case with any multi-view tool. Secondary notation capability in DPTool is poor, but has been addressed through the enhanced annotation capability in MaramaDPTool.

DISCUSSION AND FUTURE WORK

As described in the introduction, we feel that in addition to DPML (particularly the DPML meta model) and the prototype tool support we have developed for it, our most significant contributions have been in the area of dimensions and instantiation of patterns into designs. These both have some novel characteristics which have more general applicability. Accordingly it is instructive to understand how we developed these concepts in comparison to alternatives.

In the DPML metamodel we wanted a concept that would allow us to create models including groups of objects of arbitrary size. There were various other approaches we could have taken besides using the Dimension concept. One simple approach would have been to allow the cardinality of a Participant to be specified directly, indicating how many objects it represents; this is the approach taken in RBML (Kim *et al*, 2003). This would have allowed groups of objects to be detailed by a single Participant but lacked a certain expressive power. One could not, for example, express the fact that a Participant represents objects that can be classified into sets by multiple criteria, that it effectively has sets of sets of objects. Another method we considered was the set-based approach of LePUS. This would have allowed us to create a set of objects and then a set of sets of objects etc. The main drawback is that it implies an unnecessary ordering on the groupings of the objects, you must group them into sets according to some criteria first and then group the sets into sets according to a second criteria and so on. The order in which you apply the criteria is arbitrary but fixed, once specified you cannot go back and re-order the groupings to suit the different relationships the groupings may be involved with.

We came up with the concept of a Dimension to get around this problem. Designers can specify that a Participant has a certain number of dimensions but no ordering of those Dimensions is (nor should be) implied. The objects linked to a Participant then can be classified according to their position within a particular Dimension and we can consider the Dimensions in any order we wish, each order creating a different sequence of classifications. No order then is implied by the specification of the Dimensions, simply that this Participant has another Dimension in which the objects attached to it can (and indeed must) take up a position. Another advantage of the Dimension concept is that the same Dimension can be applied to different Participants to imply they have a similar cardinality in that one direction. This enables us to easily specify constraints, such as two Participants have exactly the same cardinality, by giving them both the same Dimension. Also if one Participant has a Dimension and another Participant has that Dimension plus a second Dimension then we are saying the second Participant represents a group of objects for every object in the first Participant.

The proxy elements were also an interesting design decision. We considered, initially, replicating a Design Pattern's structure in a Design Pattern instance by simply linking the original objects from the Design Pattern into the instance. However this technique would result in an unnecessarily messy and inelegant object structure in our DPML models. The same object would have been linked into many instances and had potentially many different instance-specific alterations added to it. The proxy elements allow us to maintain a certain separation between the instances and the original pattern while still having a mechanism for keeping the structures consistent. It is not possible in the metamodel to

express the fact that, in an implementation of the DPML, the proxy elements should listen to their base element and make changes, when required, to maintain consistency with that base element. However you can specify that, in a correct model, proxy elements must be consistent with their base elements. The proxy elements then take part in all the relationships and activities that have instance-wide scope, while alterations and relationships that have pattern-wide scope take place in the pattern and, in a tool implementing the DPML, can be propagated to the instances via the proxy elements. Proxy elements help us maintain the self-contained nature of the original design pattern models so that they can be used independently from models of their instances (although not vice versa!).

In both cases, dimensions and proxies, we feel that the approach we have taken is more generally applicable, in particular to any model driven development situation where an element in the model can represent and be instantiated as a multiply categorised set and where models have multiple, tailored instantiations respectively. We are, for example, exploring the use of dimensions in our meta tool model specification languages as an alternative to the cardinality approaches we are currently using. The decisions about what object structures to model as Participants and what to model by using Constraints or Relations were the most difficult ones we had. The set of Participants we came up with was fairly standard by most measures. However arguments can be made for modelling an Attribute as a Constraint on an Implementation, rather than as a Participant as we have done, or for modelling constructs such as Associations as Participants, rather than with a BinaryDirectedRelation, which is the approach we took. We were largely influenced in our decisions by what structures in an OO design have a clear definition in an implementation (such as a class, a method or an attribute) because these fell naturally into roles as Participants. If associations between classes in a mainstream programming language are ever given a clearer, more encapsulated definition than as a pair of object references, our perception of associations could well change. This would be an argument for having Associations as Participants rather than as a condition that exists between two other Participants.

Another area that requires further work is pattern composition. It is obvious after working with DPML for a while that some combinations of Participants or mini-patterns are very common and are repeatedly re-used in design after design. This was an aspect also noted by our survey respondents. While our model and tool support instantiation of multiple patterns into a UML model, there is no support for composing patterns from other patterns. In the DPML metamodel the direct superclass of DesignPattern and DesignPatternInstance, DesignPatternNamespace, is a direct subclass of DesignPatternModelElement. We gave some consideration to making DesignPattern a subclass of Participant to facilitate a form of design pattern composition. With the ability to include a whole DesignPattern as a Participant of another pattern you could specify a simple mini-pattern in a DesignPattern and use it in many other patterns, even with Dimensions to specify multiple groups of the DesignPattern. There are complicating issues that arise with this addition to the metamodel however, particularly in the changes required to the instantiation metamodel and in the semantics of linking additional Constraints to the Participants involved in the sub-pattern, which leads to the whole notion of visibility of Participants from outside the DesignPattern. We have not been able to address these issues satisfactorily yet and so we have not as yet been able to incorporate this idea for pattern composition into our metamodel or tool support. By contrast, Mak *et al* (2003) have developed an extension to LePUS (exLePLUS) which provides pattern composition capability. Noble and Biddle (2002) provide a deeper discussion of the relationships that exist between patterns that provides a useful basis for further developing pattern composition tool support.

Generalising, it can be seen that other, e.g. more domain specific, participants could be selected creating a variety of DPML-like variants for use in model driven development. Our metamodel is readily adaptable to this approach, and the meta-toolset based implementations we have been using for implementation likewise allow for rapid adaptation to incorporate new participant types and related icons. We thus see our developing MaramaDPTool as an underlying framework for rapidly developing domain specific modelling languages that support instantiation into UML designs as part of an overall model driven development approach. In this respect we have similar aims to those behind the development of Microsoft's DSLTool meta toolset (Microsoft, 2006). This type of approach is likely to suit constrained situations, such as product line configuration, where domain specific visual notations afford a more accessible approach for end user configuration than existing conventional coding approaches.

CONCLUSIONS

We have described DPML and associated toolset for specifying and instantiating design patterns. Patterns specified using DPML can be instantiated into UML designs. The instantiation process supports customisation, to adapt the pattern instance for the particular context it is applied to. Two proof-of-concept tools have been developed supporting the use of DPML. Evaluations, both user and cognitive dimensions based, demonstrate the usefulness and effectiveness of the language and tool support. Of particular novelty is our approach to specifying multiplicity of participants in a design pattern, using the dimension concept, combined with binary extended relations that interpret the dimensions in a particular context. We feel these have broader applicability in other areas of model driven development.

ACKNOWLEDGEMENTS

Support for this research from the New Zealand Public Good Science Fund and the New Economy Research Fund is gratefully acknowledged. David Mapelsden was supported by a William Georgetti Scholarship.

REFERENCES

- Chambers, C., Harrison, B., & Vlissides, J. (2000). A debate on language and tool support for design patterns, In Wegman M. & Reps, T. (Eds) *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, (pp 277 – 289) Boston ACM Press.
- Coplien, J.O. (1996). *Software Patterns*. SIGS Management Briefings. SIGS Press.
- Eclipse Foundation (2006), Eclipse, retrieved 30 March, 2006 from <http://www.eclipse.org/>
- Eden, AH. (2002). A visual formalism for object-oriented architecture", In *Proc Integrated Design and Process Technology*, Pasadena, California.
- Florijn, G., Meijers, M., & van Winsen, P. (1997). Tool support for object-oriented patterns", in *ECOOP '97 – Proceedings of the 11th European conference on Object Oriented programming*, LNCS 1241, 472-495
- Fontoura, M., Pree, W., & Rumpe, B. (2002). *The UML Profile for Framework Architectures*. Addison-Wesley.
- France, R.B.; Kim, D.-K.; Ghosh, S., & Song, E. (2004). A UML-based pattern specification technique, *IEEE Trans SE* 30(3), 193-206.
- Gamma, E., Helm, R., Johnston, R., and Vlissides, J. (1994). *Design Patterns*, Addison-Wesley.

- Grand, M. (1998). *Design patterns and Java*, Addison-Wesley
- Green, T.R.G & Petre, M., (1996). Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework, *Journal of Visual Languages and Computing*, 7, 131-174
- Grundy, J.C., Hosking, J.G., Zhu, N., & Liu, N. (in press). Generating domain-specific visual language editors from high-level tool specifications, In *Proc 2006 ACM/IEEE Conference on Automated Software Engineering*.
- Guenec, A.L., Sunye, G., & Jezequel J. (2006). Precise modeling of design patterns. In *Proceedings of UML '00* (pp. 482-496)
- Kim, D., France, R., Ghosh S., & Song, E. (2003). A UML-Based Metamodeling Language to Specify Design Patterns, *Proceedings of Workshop on Software Model Engineering (WiSME), at UML 2003*, San Francisco.
- Lauder, A., Kent, S. (1998). Precise Visual Specification of Design Patterns, In *Proc ECOOP'98 Workshop reader on OO technology, LNCS 1445*, pg114-134.
- Mak, J.K.H., Choy, C.S.T., & Lun, D.P.K (2003). Precise specification to compound patterns with ExLePUS, In *Proc. 27th Annual International Computer Software and Applications Conference*, 440-445
- Mak, J.K., Choy, C.S.T., & Lun, D.P.K. (2004). Precise Modeling of Design Patterns in UML, In *26th International Conference on Software Engineering (ICSE'04)*, 252-261.
- Mapelsden, D., Hosking, J.G., & Grundy, J.C. (2002). Design Pattern Modelling and Instantiation using DPML, In *Proc. Tools Pacific 2002*, Sydney, IEEE CS Press
- Microsoft (2006). *Domain-Specific Language Tools*. retrieved 30 March, 2006 from <http://msdn.microsoft.com/vstudio/DSLTools/>
- Noble, J., & Biddle, R. (2002) Patterns as Signs. In *Proceedings of the European Conference on Object Oriented Programming*, Spain. (ECOOP). pp368-391. Springer-Verlag
- Object Management Group (2006). Unified Modelling Language (UML) Specification v 2.0., retrieved 30 March, 2006 from www.omg.org/technology/documents/formal/uml.htm.
- Reiss, S.P. (2000). Working with patterns and code, In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, Volume: Abstracts*, (pp 243 –243).
- Robbins, J.E., & Redmiles, D.F. (1999). Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, In *Proc CoSET'99* (pp. 61-70) Los Angeles: University of South Australia.
- Sefika, M., Sane, A., & Campbell, R.H. (1996). Monitoring Compliance of a Software System with its High-Level Design Models. In *Proceedings of the 18th international conference on Software engineering*. Berlin, Germany. 387 – 396
- Taibi, T., & Ngo, D.C.L. (2003). Formal Specification of Design Patterns – A Balanced Approach, *Journal of Object Technology* 2(4), 127-140
- Zhu, N., Grundy, J.C., & Hosking, J.G. (2004). Pounamu: a meta-tool for multi-view visual language environment construction, In *Proc. IEEE Visual Languages and Human Centric Computing Symposium*, Tokyo, 254-256.

4

Software Development and Testing using DSLs and MDE

4.1 Supporting Multi-View Development for Mobile Applications

Barnett, S., Avazpour, I., Vasa, R., Grundy, J.C. Supporting Multi-View Development for Mobile Applications, *Journal of Computer Languages*, Volume 51, April 2019, Elsevier, Pages 88-96

DOI: [10.1016/j.cola.2019.02.001](https://doi.org/10.1016/j.cola.2019.02.001)

Abstract: Interest in mobile application development has significantly increased. The need for rapid, iterative development coupled with the diversity of platforms, technologies and frameworks impacts on the productivity of developers. In this paper we propose a new approach and tool support, Rapid Application Tool (RAPPT), that enables rapid development of mobile applications. It employs Domain Specific Visual Languages and Modeling techniques to help developers define the characteristics of their applications using high level visual notations. Our approach also provides multiple views of the application to help developers have a better understanding of the different aspects of their application. Our user evaluation of RAPPT demonstrates positive feedback ranging from expert to novice developers.

My contribution: Co-developed main ideas for the research, co-supervised PhD student, wrote substantial part of the paper

Supporting Multi-View Development for Mobile Applications

Scott Barnett^a, Iman Avazpour^a, Rajesh Vasa^a, John Grundy^b

^a*Deakin Software and Technology Innovation Lab (DSTIL), School of Information Technology, Faculty of Science, Engineering and Built Environment, Deakin University, Burwood, Victoria 3125, Australia*

^b*Faculty of Information Technology, Monash University, Monash, Victoria 3800, Australia*

Abstract

Interest in mobile application development has significantly increased. The need for rapid, iterative development coupled with the diversity of platforms, technologies and frameworks impacts on the productivity of developers. In this paper we propose a new approach and tool support, Rapid APPLication Tool (RAPPT), that enables rapid development of mobile applications. It employs Domain Specific Visual Languages and Modeling techniques to help developers define the characteristics of their applications using high level visual notations. Our approach also provides multiple views of the application to help developers have a better understanding of the different aspects of their application. Our user evaluation of RAPPT demonstrates positive feedback ranging from expert to novice developers.

Keywords: Visual Notation, Mobile App Development, Domain Specific Languages, Code Generation

1. Introduction

Mobile application (app) development has exploded. In 2014 the Google Play Store doubled its number of apps¹ and in December 2014 over 40000

Email addresses: `scott.barnett@deakin.edu.au` (Scott Barnett),
`iman.avazpour@deakin.edu.au` (Iman Avazpour), `rajesh.vasa@deakin.edu.au`
(Rajesh Vasa), `john.grundy@monash.edu` (John Grundy)

¹<http://blog.appfigures.com/app-stores-growth-accelerates-in-2014/>

apps were submitted to the iTunes App Store². Despite the popularity, app development is a tedious process as it requires copious amounts of code to be written using tools that lack support for high level abstractions. Modeling techniques such as Domain Specific Visual Languages (DSVL) simplify development of mobile apps by abstracting away the details, hence improving developer productivity. On the other hand, downsides from using modeling techniques consist of rigidity in the generated output and the lack of flexibility for specifying custom functionality. Mobile app development follows a user-centered design process where real-world human experience is central to the design process [1]. Hence, developers must focus on building a great user experience and cannot afford to be limited by the restrictions of a model based tool.

Typically an IDE consists of a code view and a designer for specifying the UI components of a screen. From these views developers cannot easily grasp the navigation flow of the app, identify where the resources provided by a single API are used, or clearly analyze the event handling for a single screen. The information for each of these concerns is embedded in the source code requiring developers to browse through the code. This lack of transparency restricts communication as not all stakeholders understand code.

Early high fidelity prototyping helps with communication of ideas. Due to the user centered aspect of mobile app development many developers choose to prototype. These prototypes often focus on the visual aspects of the mobile app striving to resemble the final app as closely as possible. Once the visual aspects of an app have been agreed upon these prototypes are discarded requiring developers to replicate the prototype in code. Prototyping other non-visual aspects of an app is rarely done due to the cost.

To aide professional app developers, this paper presents our approach in Rapid APPlication Tool (RAPPT) [2], a framework for multi-view development of mobile apps. RAPPT leverages model driven techniques to bootstrap mobile app development through use of a Domain Specific Textual Language (DSTL) and a Domain Specific Visual Language. It provides multiple views to developers ranging from detailed view (code) to abstract view (e.g. page navigation). These multiple views help developers implement mobile app concepts at different levels of abstraction improving the customizability of

²<http://www.statista.com/statistics/258160/number-of-new-apps-submitted-to-the-itunes-store-per-month/>

the app. These multiple views also improve communication between stakeholders by presenting a snapshot of the application for a specific concern. Key contributions of this paper are:

- A classification of the different levels of abstraction present for mobile app development.
- A multi view system is introduced that enables developers to work on different abstraction levels synchronously.
- We provide tool support for the fast development of mobile app prototypes using a multi-view modeling approach.

This paper is organized as follows: Section 2 provides a motivating scenario for this work. Next, we present the related work in Section 3. In Section 4 we provide background information on the different abstractions used in the development of mobile apps. Section 5 describes our approach including the various elements of the framework and concepts in the DSL. Following that is an overview on our tool implementation RAPPT in Section 6. Section 7 provides details of our user evaluation and experiment setup and finally Section 8 concludes the paper and provides avenues of future work.

2. Motivation

Our primary motivation for this research comes from experience with developing mobile apps. In particular, dealing with the inadequate tools provided to professional developers when starting a new app. As a way to illustrate the mobile app development process and the tools used we present a motivating example for the development of a data-intensive mobile app, MovieDBApp. A data-intensive mobile app is an app that is predominantly focused on fetching, formatting and rendering data retrieved from a web-service.

Consider Peter, a mobile application developer, who is tasked with the development of MovieDBApp. MovieDBApp shows a list of popular movies that when selected enable a user to navigate to a screen showing the details of that movie. This app consists of three screens a *Popular Movies* screen for displaying the list of popular movies, an *About* screen to display copyright information, and a *Movie Detail* screen to display the details of a selected movie. Screenshots for these screens are shown in Figure 1. The content to

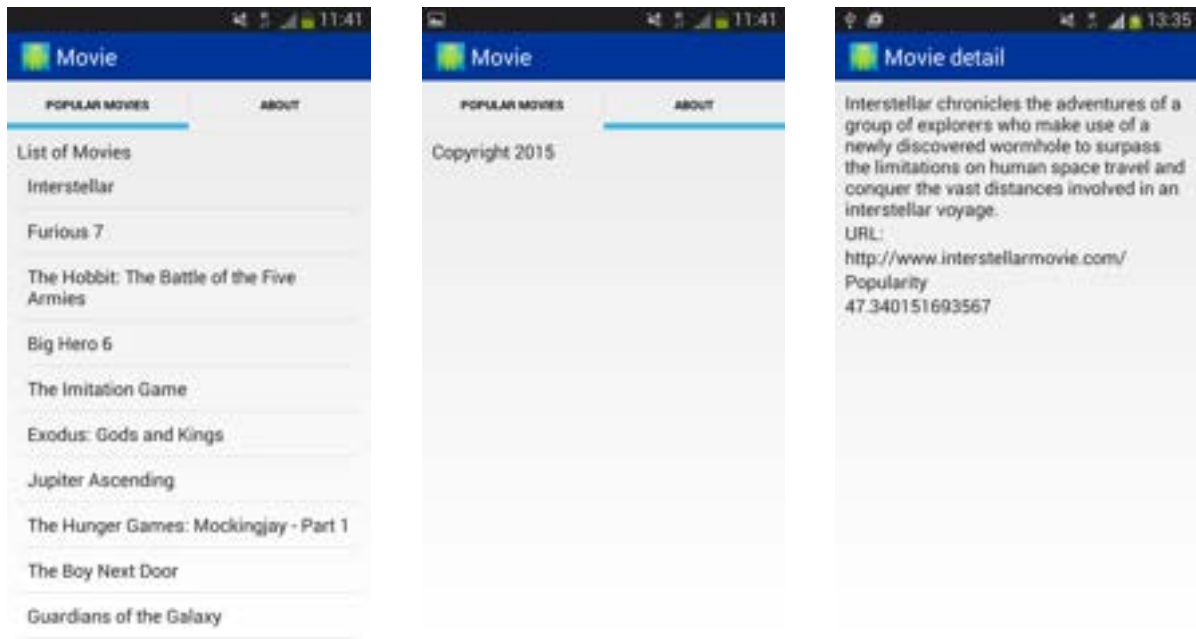


Figure 1: The *Popular Movies*, *About*, and *Details* screens for MovieDB, an app based on the MovieDB API.

be displayed by this app will be provided by the freely accessible MovieDB API³. Peter's app includes the following requirements:

1. A tabbar for navigation between the *Popular Movies* and *About* screens. Tabbar is a Mobile UI Pattern that displays tabs near the top of the screen for navigation (see Figure 1). The user can also swipe the screen to navigate between tabs.
2. Authentication for connecting to the Movie DB API e.g specify an authentication key.
3. Display a list of the popular movies from the Movie DB API on the *Popular Movies* screen.
4. Navigate to the *Details* screen from the *Popular Movies* screen by selecting a list item.
5. Pass the identifier for a movie when clicking on a list item in the *Popular Movies* screen and pass it to the *Details* screen so the details for the correct movie can be fetched.

³<http://docs.themoviedb.apiary.io/>

6. Fetch the details for a selected movie from the MovieDB API and render the results to the screen on the *Details* screen.
7. Display a copyright message on the *About* screen.

To build this app, Peter needs to write the code for the tabbar specifying the tabs for each screen, configure the navigation for each tab, handle navigation to the tab screen and create the animation for swiping between tabs. In order to connect to the MovieDB API and handle authentication, he must write code for the API requests, model the data returned by API calls, handle errors, check network connectivity, authenticate with the MovieDB API and ensure best practices for concurrency on a mobile platform. For the *Popular Movies* screen, Peter needs to create the following: layout files for the UI, connection between data returned from API call and the UI components, event handlers for selecting a list item to navigate to the *Details* screen, and implement navigation pattern tabbar. The *Details* screen needs to accept the parameters passed from the *Popular Movies* screen, pass that parameter to an API call to fetch that data for the selected movie and render the movie details to the screen. The *About* screen needs to display a static message to the user. In addition to these tasks Peter needs to configure the build system, adding any dependencies, add logging code, create styles, debug, follow software engineering best practices such as ensuring maintainability and performance, while meeting stringent deadlines.

3. Related Work

Many of the current tools are aimed at inexperienced developers or non-technical experts [3, 4, 5, 6, 7, 8]. These approaches hide many of the implementation details described above from the end users [9] enabling them to focus on higher level constructs. While this makes programming more palatable, end users cannot customize how these apps address the intricate concerns of developing mobile apps [10, 11, 12, 13]. For example, Peter has specific requirements about where to fetch data from (i.e. the MovieDB API) and how to authenticate the app with the API using an API key. As such these approaches are not suited for use by a professional mobile app developer.

Multiple Model Driven Development (MDD) approaches for building mobile apps have been developed [9, 14, 15, 16, 17, 18]. These approaches commonly focus on MDD as a cross-platform approach to mobile app development. A common emphasis of these cross-platform approaches is on

modeling a common subset of features that can then generate code for multiple target platforms. The modeling approaches often present a single way of modeling a mobile app and do not provide multiple views of a mobile app as in our proposed solution. For example, describe every aspect of an app by using a DSL [14]. In addition these MDD approaches tradeoff flexibility for productivity by abstracting away many of the implementation details. Data-intensive mobile apps share a lot of functionality yet have very unique UI and interactions. To support building unique apps a tool should support a high degree of customization. Recent work has also focused on using MDD for building context aware applications [19, 20]. These approaches focus on a subset of the functionality available for mobile apps where as RAPPT is designed around building data intensive applications.

Typically the app development process begins by designing the UI and User Experience with the client before these ideas are refined into mock-ups by a designer. Tools for prototyping [21, 22, 23] can greatly help with the evaluation of ideas especially concerning the navigation flow through an app. Once this process has been done developers have to start from scratch to implement the agreed upon navigation flow – current tools produce throwaway prototypes. In addition, implementing a prototype that can be reused by developers is a time intensive and costly process. What is needed is a tool that can be used for rapid prototyping but generates apps that can be built upon by developers when realizing the final app.

User Experience plays a crucial role in the development of mobile apps due to the influence of user reviews and rankings of the app store. Designers and developers need to pay careful attention to the usability of their apps. Modeling different components of an app such as the navigation flow is well suited to and frequently represented as a graphical visualization. Developers often spend a lot of time programming using text and is so more familiar to them for describing event handlers, data flows and UI bindings. Researchers have found multi-view approaches to be beneficial for addressing multiple concerns when building multiagent systems [24], in embedded systems engineering [25], and in the design of cyber-physical systems [26]. This shows that a multi-view approach is suitable for addressing the specific concerns of mobile app development [11, 12].

From our motivating example and review of associated literature we have identified the following key requirements for a tool to assist mobile app developers that should:

- R1. Automate generation of the boilerplate code required for data-intensive mobile apps.
- R2. Design a tool for professional mobile app developers.
- R3. Generate apps that provide the full capabilities of the underlying platform i.e. provide flexibility in the generated apps.
- R4. Enable rapid prototyping of a fully functioning mobile app.
- R5. Produce prototypes that can be refined into the final app.
- R6. Provide multiple abstraction levels for modeling the different concerns of a mobile app i.e. navigation flow and UI composition.

4. Mobile App Concepts

App development involves different levels of abstractions that build upon the previous level and allow developers to reason about different aspects of a mobile app. Building tools at the appropriate level of abstraction requires making trade-offs between flexibility and productivity as the higher levels are harder to modify. Each level of abstraction is discussed below from the lowest level to the highest.

- *Low Level Platform APIs.* This is the lowest level of abstraction provided by a mobile platform. Being the lowest level, it permits the maximum flexibility as developers can modify every aspect of the platform. Typically the rest of the platform's APIs are implemented using this low level API.
- *Base Components and APIs.* Platform vendors provide components from which app developers build their apps such as buttons, classes for creating screens and fetching data from webservice or for interacting with platform features such as sensors. The majority of a mobile app is currently developed at this level of abstraction as it strikes the appropriate trade-off between flexibility and productivity.
- *High Level App Concepts.* These are the concepts that non-developers can reason about or end users can experience. Implementing these concepts needs significant development work, requiring the use of multiple

lower level APIs. Example concepts at this level are screen, call to webservice, maps etc. Current literature on DSLs for the mobile app domain focus on capturing the abstractions present at this level but lack the capabilities for the lower level abstractions prohibiting flexibility in created apps, or fail to provide higher level views of the entire app to developers. The concepts at this level can often be mapped across platforms.

- *App Concerns*. This is the highest level of abstraction and is focused on viewing and manipulating app wide concerns such as navigation flow and data modeling. Model driven engineering approaches to mobile app development often work at this level of abstraction.

Mobile app developers have to address all four levels of abstraction and work in a rapidly changing environment where high level abstractions have yet to be defined for new app features – new features do not have stable or well defined abstractions suitable for model driven approaches. For example, mobile platform vendors frequently update their design guidelines but may not update their APIs at the same time requiring developers to build custom components from scratch. Consider the example of using a UML tool (App Concerns) [27] to model a data-intensive mobile app which includes a new UI navigation pattern such as a Drawer ⁴. Implementing a Drawer requires using High Level App Concept of a list, Base Components and APIs for displaying text and images, and Low Level Platform APIs to handle custom animation and styles. Developers cannot always wait for the platform to implement a feature they need or for a 3rd party library to become available. Essentially all 4 levels of abstraction have to be considered as first class citizens when building a model driven tool for the mobile app domain due a rapidly changing environment.

5. Our Approach

Our approach RAPPT is a tool that provides multiple views for specifying the concepts of a mobile app and generates working prototypes that form the scaffolding of the final app. The focus of our work has been on building a tool that targets professional app developers and assists them in the early

⁴<https://www.google.com/design/spec/patterns/navigation-drawer.html>

stages of development by providing them with a DSL to specify high level app features and then add extra details using a DSTL. After the specification has been completed RAPPT generates the source code for a single platform, a working Android app, to which the developer adds the final polish. The major steps involved in using RAPPT to generate a new project are shown in Figure 2. The core aspects of our approach *Generate an App for a Single Platform*, use an *Empirically Derived Meta-model* and use *Multiple Views with Overlapping Abstraction Levels* will be described in detail below.

5.1. How RAPPT Works

This section outlines how developers use RAPPT to build a mobile app and the numbers below map to Figure 2.

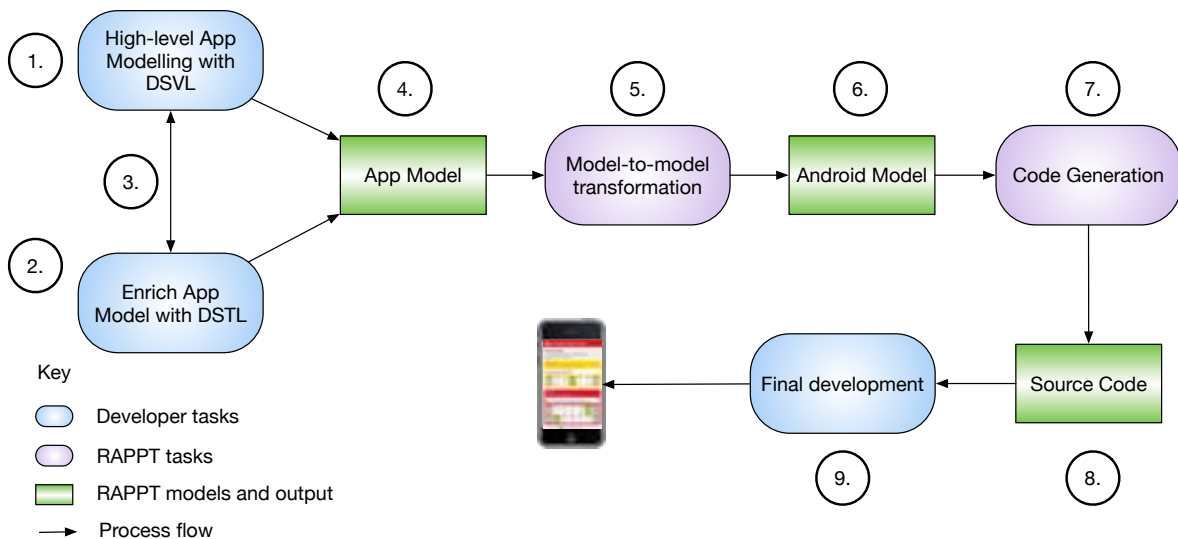


Figure 2: RAPPT assists software developers by generating the initial architecture for their app.

1. Developers start using RAPPT by describing the high level structure of an app using the DSVL. This includes specifying the number of screens in the app, the navigation flow between screens and major features per screen such as Google maps.
2. Developers can then switch to using the DSTL to provide additional details to the mobile app that are not provided by the DSVL. These features include specifying details about a web service for fetching data,

defining the data schema, specifying and configuring authentication, and specifying the information to display in specific UI elements such as rows in a list.

3. Both the DSVL and DSTL update an App Model permitting developers to switch between the interfaces as they proceed through the modelling process. Developers can also view the source code that will be generated from their model.
4. RAPPT then performs a model-to-model transformation to convert the App Model to an Android Model. This process fills in additional details about the Android app to ensure that the generated app closely resembles that of written by a developer. These features include adding the correct permissions, selecting the correct classes from the Android SDK i.e. Fragment vs Activity, externalising strings for internationalisation and adding the appropriate dependencies to the project.
5. The Android Model contains all of the information required for generating the source code. Future work can explore using this model to identify violations of mobile app development best practices. The Android Model also contained fields that were not to be used in the generated code such as the directory structure following Android conventions.
6. Next, RAPPT maps code templates to the Android Model and generates source code.
7. The generated source code from RAPPT contains the scaffolding for the mobile app that is ready to be run on the device. This code can be compiled and tested immediately after generation.
8. Developers are needed to finish off the generated app by adding business logic and styling.

5.2. *Generate an App for a Single Platform*

One of the key motivations of our approach is requirement *R3. Generate apps that provide the full capabilities of the underlying platform i.e. provide flexibility in the generated apps.* To achieve this we focused on generating code for a single platform which meant we did not need to handle discrepancies between platforms and could generate code that adhered to the

platform’s UI guidelines. We designed our code generator to produce mobile apps that could be compiled and deployed to a device without modification satisfying the requirement of *R4. Enable rapid prototyping of a fully functioning mobile app.* This enables developers to produce the first prototype quickly and can gather feedback on the navigation flow for the app during the initial client meeting. As mentioned above app development begins with the UI and UX and then moves onto the development tasks. To enable a smooth transition from the prototyping stage and to ensure there is no wasted effort we designed the generated apps in a way that it forms the scaffolding for the final app. Once the initial prototyping stage was complete developers take the generated app and build the rest of the app on top of what was generated enabling RAPPT to satisfy requirement *R5. Produce prototypes that can be refined into the final app.*

5.3. Empirically Derived Meta-model

Our approach leverages techniques from Model Driven Development (MDD) especially the field of Domain Specific Languages in the generation of mobile apps. MDD’s use of models to abstract away implementation details is well suited to address the requirement *R1. Automate the boilerplate code required for data-intensive mobile apps.* Underpinning both our DSVL and DSTL is a shared meta-model, shown in Figure 3, for data-intensive mobile apps that contains the core concepts required to model and generate a working mobile app. It is the shared meta-model that enables the user to switch between editing the DSVL and the DSTL, and that provides developers with the productivity boost by providing them with high level abstractions.

There are two main reasons why we decided to derive a meta-model. First, we wanted to identify the smallest number of concepts that can be mapped to source code that addresses the technical domain concerns of mobile apps [13]. Second, our focus was on identifying concepts used frequently in real apps rather than assuming all concepts offered by a mobile app SDK are essential for rapid generation.

All of the abstractions are available in the DSTL but only some of the concepts are available in the DSVL as they are not all needed to specify the high level concerns of a mobile app. As shown in Figure 3, the concepts of web services, data fields and the data model can only be specified by the DSTL due to the requirement to accept input from the user i.e the URL for the web service requires text input. The requirement *R2. Design a tool*

aimed at professional mobile app developers, meant that the meta-model also had to include concepts that developers use to describe mobile apps.

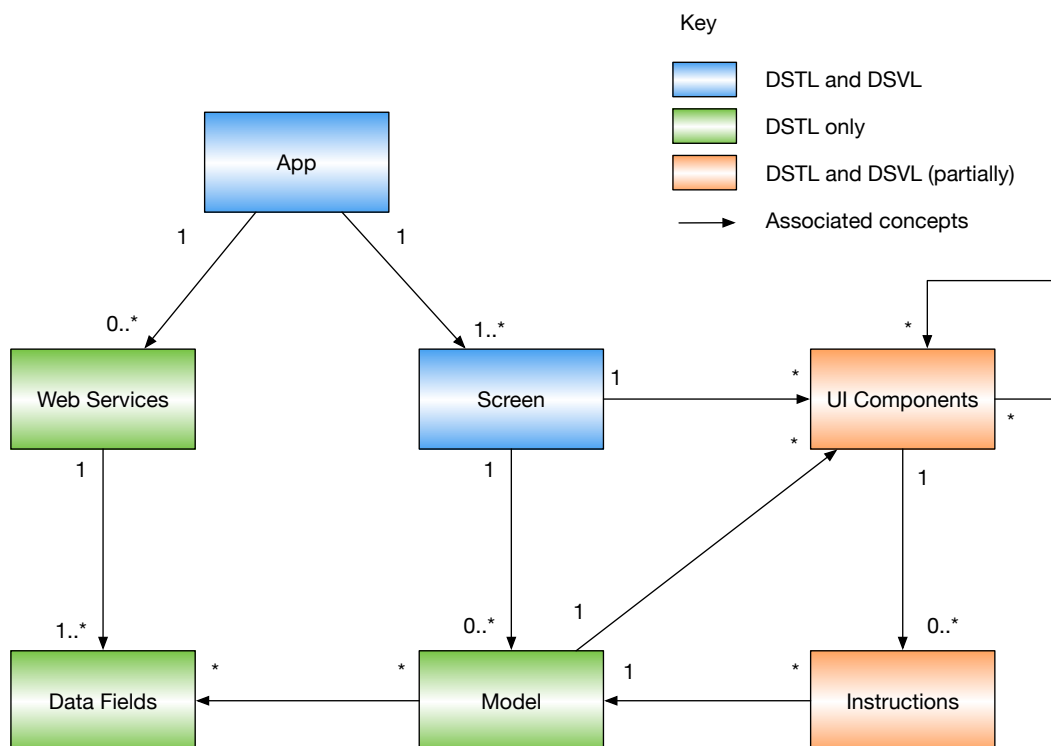


Figure 3: RAPPT's meta-model underpinning the DSVL and DSTL. Coloured boxes show which concepts are available in the DSVL and DSTL.

Each app has specific settings unique to that app such as the first screen is to be shown when the app launches. These concerns are captured in our meta-model by the concept of *App*. In order for an *App* to be able to model data-intensive mobile apps, it should include two major concepts *Web Services* which provide the contents for the app, and *Screens* that display the data to the end user. *Data Fields* represent the data that is returned from a *Web Service* and forms the data model of the mobile app. The data model for an app *Model* captures the structure of the data and what is shown to the user. *Screens* are made up of *UI Components* that display parts of the data model and both may trigger an *Event*. *Events* may be fired from a button or be triggered with the screen loads and contain *Instructions* which perform a task; most commonly a call to a web service to fetch data to display or render to the screen.

To create the meta-model we analyzed 30 data-intensive mobile apps and

from these extracted the core concepts. A data-intensive app was considered an app that 1) primarily relied on data to provide its functionality, 2) was a complete app rather than a live widget running on the home screen and 3) was not categorized as Games. Studying these apps, we built a primitive version of the meta-model which we used to generate new apps. When we identified concepts that the tool could not generate we added the concepts to the meta-model and then extended RAPPT. In this manner, we were able to follow an iterative process to build the meta-model. These concepts formed the basis for the concepts in our DSTL that we have described previously [2] and informed the design of our DSVL. List of the studied apps is available online⁵.

5.4. Multiple Views with Overlapping Abstraction Levels

Software engineers utilize higher level abstractions to hide the unnecessary details and hence focus on the problems at hand. As such we needed to ensure that RAPPT could satisfy requirement *R6. Provide multiple abstraction levels for modeling the different concerns of a mobile app i.e. navigation flow and UI composition..* For each of the views discussed in Section 4, different abstractions are needed. These abstractions could conflict with each other. Navigation and UI layout views both need a screen but require different information. The navigation view is concerned with how a screen is connected to other screens whereas the UI layout view provides precise configuration of UI components.

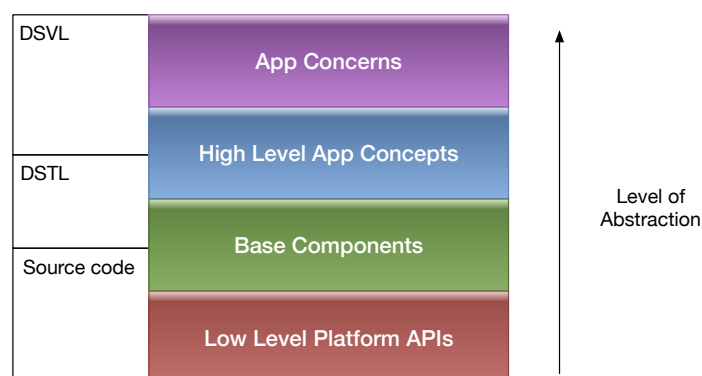


Figure 4: Levels of abstraction in a mobile app.

⁵<https://github.com/ScottyB/analysed-apps>

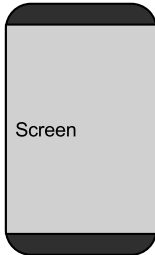
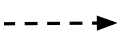


Our approach provides three views for developing mobile apps consisting of overlapping levels of abstraction to address these conflicting scenarios: A DSVL for high level app functionality, a DSTL for providing additional details not available in the DSVL and access to the target platform for creating custom app functionality. Figure 4 provides an over view of these abstraction levels, their composing elements, and their relations.

Underpinning both the DSTL and the DSVL is a meta-model of an Android app that is used to generate code once the initial modeling stage is complete. The DSVL includes concepts for modeling the high level concepts (*App Concerns*) of data-intensive apps such as navigation flow and the Data Model. Included in the DSVL are a number abstractions that are categorized as *High Level App Concepts*, and can be enhanced using the DSTL. For example, the concept of a screen is a *High Level App Concepts* that is present in the DSVL so that the navigation flow can be modeled but the DSTL is required to specify what will be displayed on the screen or which webservices it calls. A summary of the concepts present in the Visual Language is shown in Table 1. The rationale for choosing visual notations was to choose the representations that closely relate to the concepts in the meta model and the target platform (here Android operating system). Only a few concepts have been added to the Visual Language as the focus of this paper is on the multi-view approach to mobile app development, rather than the complete visual app builder.

The DSTL also includes abstractions from the *Base Components* category. These abstractions are added to enhance the specification for screens and to be able to model webservices for fetching data to render to the screen. Examples in the DSTL are input fields, images and keywords for specifying webservices. For highly custom features and concepts that cannot be modeled by any of the previous stages developers have to use the *Low Level Platform APIs* in with the generated code. Once the developer has finished modeling RAPPT generates code for the developer to modify which provides maximum flexibility to the developer as they are only limited by what the target platform provides.

⁶<https://developers.google.com/maps/documentation/android/>

Table 1: Example visual elements that make up RAPPT’s Visual Language

Concept	Notation	Description
Screen		Represents a screen displayed on a mobile device as seen by the end user.
Button Navigation		Represents navigation from one screen to another by clicking on the UI component Button.
Map		Displays a Google Map ⁶
Tabbar		Represents the Mobile navigation UI pattern, Tabbar.

5.5. DSTL Grammar and Syntax

The grammar for the DSTL is derived from the meta-model and the full definition is available online⁷. Both the app and screen concepts from the meta-model have a direct mapping to concepts in the DSTL. All of the other concepts in the meta-model represent an abstract instance of the entities in the DSTL. For example, Web Services in the meta-model includes the following abstractions *api* for specifying a datasource, *api-key* for authentication, and *GET,POST* for making calls to a webservice. Example code from the DSTL used in RAPPT can be seen in Figure 5.

6. Implementation

Developers interact with RAPPT through a web interface implemented using standard web technologies (i.e. HTML 5, CSS and Javascript). The

⁷<https://github.com/ScottyB/rappt/blob/master/grammar.g4>

```

screen MovieDetailScreen "Movie detail" {
  group movieDetailGroup {
    ① on-load {
      ② call MovieDB.movieDetail passed idParam
    }
    image backdropId backdrop_path:image ③
    label title2ID title
    image posterImageId poster_path:image
    label overViewId overview
    label popularityLabel "Popularity:"
    label restPopularity popularity
  }
}

app { ④
  landing-page MoviesScreen
}

api MovieDB "https://api.themoviedb.org/3" {
  ④ api-key api_key "<API KEY>"
  GET popularMovies "/movie/popular"
  GET movieDetail "/movie/{id}"
}

```

Figure 5: Sample DSTL code for loading data from an API and showing the results to the screen: 1) an event handler for catching the on screen load event, 2) a call to an API with parameters, 3) the landing page for the generated app and 4) definition of the API.

App Model was implemented as a JSON object and was shared between the server and the client. Using JSON as the model format meant that it could easily be sent to and from the server as many web apps use JSON as a data transmission format. The DSTL Processor was implemented using the ANTLR compiler compiler which compiled the DSTL into an internal representation of the App Model. Errors that were generated on the server were sent to the client and displayed to the user. We implemented the server side code in Java and the code templates using String Template. Additional details about the tool implementation can be found online ⁸.

We made the decision to build RAPPT as a web based tool to simplify the process of getting started i.e. there is nothing to download and setup to use the tool. This also meant that the tool was available anywhere for rapid prototyping. The DSTL Processor had to be developed on the serverside due to the available libraries for a compiler compiler.

⁸<https://github.com/ScottyB/rappt/blob/master/rappt-tool-guide.pdf>

7. User Evaluation

We have conducted a user evaluation of RAPPT for mobile application development. Our primary aims in this user evaluation was to evaluate user acceptance and examine how using RAPPT can speed up application development for experienced software developers both with and without mobile application development experience. All of the resources for the evaluation task, survey questions and results are available online ⁹. The details of this user study follows.

7.1. Experiment setup and tasks

Participants were first asked to complete a demographic survey before starting the experiment. They then watched instructional videos demonstrating how to use the online editor and visual model. These learning videos also showed how to construct a small Android app using both the visual editor and AML. By using learning videos we were able to reduce bias and run multiple sessions with different participants. On completion of the learning videos participants were asked to fill out a survey to ascertain their confidence level with building Android apps with RAPPT. The next stage involved an evaluation task where participants were asked to build three screens for an app that displayed data from the MovieDB API similar to the motivation example of section 2. Participants used their own computers and accessed RAPPT online.

Participants were encouraged to use the provided samples. Once participants felt that they had completed the task, their program was downloaded and run on instructors' machine. Apps created by the participants were then installed on a device and run to ensure that there were no runtime issues caused by the generated code. It would also allow participants to see the application they have developed being instantly installed and used on a mobile device.

Upon finishing the experiment, a matching questionnaire was handed to each participant. This questionnaire was composed of 16 questions with 5 point Likert scale ranging from *Strongly Disagree* to *Strongly Agree*, and 9 open-ended questions capturing their experience of using RAPPT to build Android apps. We followed the positive questionnaire design approach as

⁹<https://github.com/ScottyB/rappt-eval/tree/master/user-eval>

suggested by Sauro et al. [28]. This would help us analyze the results faster, avoid accidental mistakes and have a more consistent set of questions.

7.2. Participants

The participants of this user evaluation were selected from software developers and researchers at Swinburne University of Technology (Australia). Overall 20 participants were recruited (17 male, 3 female). Our demographics questionnaire included six questions to capture participants' background and their experience in mobile application development particularly experience developing apps for Android operating system. These demographics questions and participant answers are provided by Figure 6.

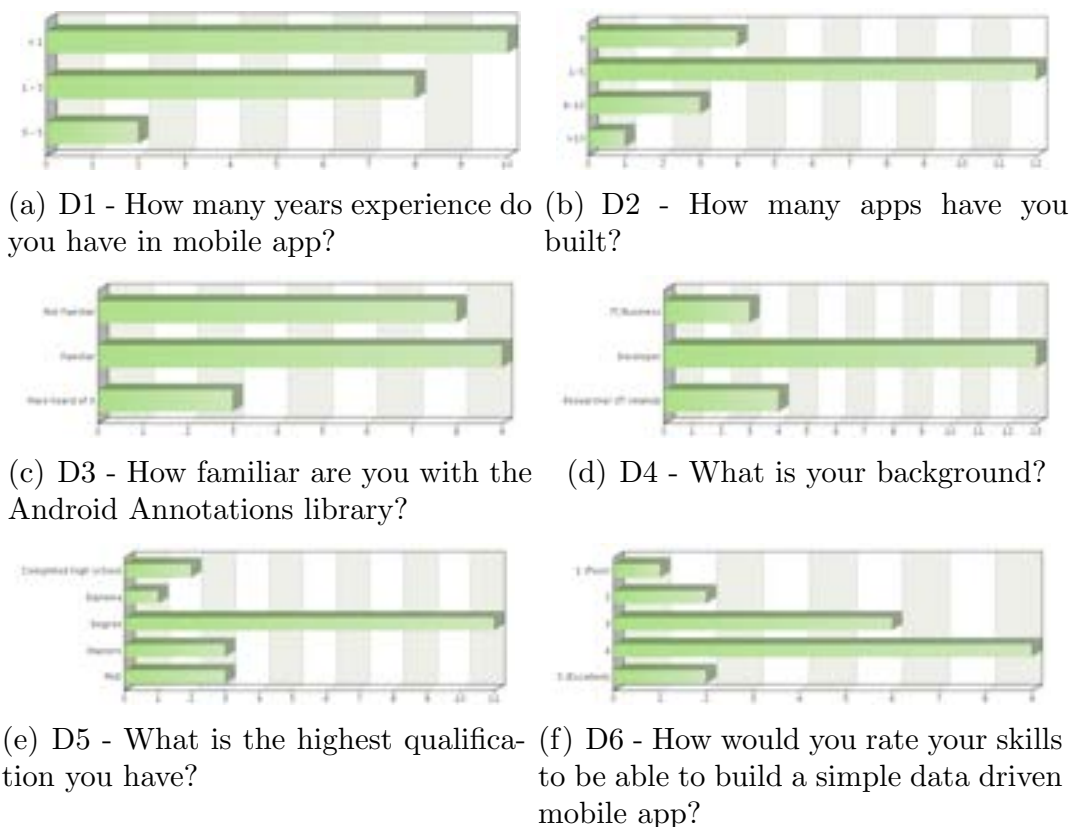


Figure 6: User demographic questions and participants' responses.

7.3. Results and discussion

Out of 20 participants, only one was *not* able to finish the experiment successfully. This was due to the participant's lack of interest in mobile app development and has been reflected in the questionnaire responses. It took the participants approximately 90 minutes on average to finish the tasks and

answer the questions. Table 2 presents a selection of questions from the questionnaire. It also depicts the frequency of participant responses to each question.

The overall response was positive with 80% of participants agreeing or strongly agreeing to 7 out of the 9 questions. These results confirm that our approach is suitable for use by professional developers

Q2 received the strongest result with 65% of participants Strongly Agreeing to the icons in the DSVL being easy to understand. This confirms our current choice of icons for the concepts present in a data-intensive mobile app. We are constantly updating the DSVL based on our feedbacks. This includes updating the icons to match target platforms, and including more meta model and fine grained concepts in the DSVL.

An interesting finding from the survey was the results to question **Q4** on RAPPT enabling users avoid making mistakes. It demonstrates a close to normal-distribution spread. The answers and comments on the questionnaire indicated needs for improving error handling mechanisms as most participants had experience with major software development IDEs. Some participants mentioned that they would have preferred a more real-time error handling mechanisms. A possible way to improve the error handling is to move more concepts from the DSTL to the DSVL – visual languages are not as susceptible to developer error as textual languages are. Improving the overall robustness of the tool is another way to address the issue of poor error reporting.

Answers to **Q5** were also non-committal with 40% of participants being neutral. The concepts in RAPPT cover the main concepts required for modeling data-intensive apps but are not exhaustive. For example the concept of a navigation pattern is present although not every navigation UI pattern is supported. This is one reason why the results may not be strongly one way or another. Clarifying the question by indicating data-intensive apps rather than all mobile apps would have removed some confusion.

Participant responses to question **Q6** indicate the overall acceptance of the approach. 95% of the participants had positive views on the usefulness of the approach (60% Strongly Agree and 35% Agree). Same is true for responses to question **Q8**.

7.4. Threats to Validity

Although we have tried our best to reduce threats to validity for the experiment, there are certain threats with regards to participant affiliations

Table 2: Sample questions of the questionnaire. Likert points have been given score of 1 to 5 representing *Strongly Disagree* to *Strongly Agree*.

No.	Question	Frequency (%)				
		1	2	3	4	5
Q.1	It was easy to use RAPPT.	0	5	15	50	30
Q.2	It is easy to understand what each icon represents.	0	0	10	25	65
Q.3	It was easy to load data from an API and render it to the screen.	0	0	15	55	30
Q.4	It is easy to avoid making errors or mistakes.	5	20	35	30	10
Q.5	The concepts in RAPPT are - sufficient for modelling a mobile app.	0	5	40	35	20
Q.6	RAPPT is useful for mobile app development.	0	0	5	35	60
Q.7	Using RAPPT is more efficient than starting with a raw Android project.	5	5	10	35	45
Q.8	You are satisfied with using RAPPT.	0	0	10	40	50

and background. In this section we outline some of these threats.

Internal Validity Our participants have been recruited from software engineers and developers at Swinburne Software Innovation Laboratory. Their affiliation may have introduced bias in their responses. Also some participants knew researchers which may have had effect on their responses to the questionnaire. During the design of RAPPT, we have considered the needs of expert as well as novice mobile app developers. The majority of our participants (18 out of 20) had less than three years experience developing mobile apps, and 10 participants had less than one year developing mobile apps. This may result in some bias towards novice app developers. However, developers or students first getting started with mobile app development can benefit from a tool like RAPPT.

External Validity The User Evaluation involved 20 participants which

is sufficient for our purposes but not for statistical analysis. Evaluating RAPPT with more developers on a wide range of applications would further improve confidence that RAPPT is generally applicable for developers. Another threat to validity is best summed up with a comment from one of the participants ... I need more time to play around with RAPPT... Participants built one app taking approximately 1 hour to complete and are likely to give different responses after using RAPPT extensively. It would be interesting to see how the answers from the participants change after using RAPPT for an extended period of time on multiple projects. The participants were asked to complete one task. An improvement to the evaluation task would involve selecting a range of apps with different functionality and purposes. Evaluating RAPPT on additional apps would also help expose gaps in the model and the language which could be used to inform the implementation.

Construct Validity In our evaluation we asked the participants to build an app which the authors verified against a set criteria available online ¹⁰. The participants were not asked to import the generated projects into an IDE which would be necessary in a commercial environment. The simple task that developers were asked to complete did not require the participants to modify the generated code. Thus, not every step required for using RAPPT was evaluated. Future work would require participants to download and extend the generated output by adding an additional feature to the application.

8. Conclusion

We have introduced RAPPT, an approach and tool support for rapid design and prototyping of mobile applications. RAPPT provides a DSVL and a DSTL for mobile application development. It also utilizes multiple views and abstractions levels of mobile applications to help developers be more efficient in prototyping various apps and at the same time, have maximized customization ability. In addition, we have presented 4 levels of abstraction present when building mobile apps and cater for the different levels in our approach. We have evaluated RAPPT using a user study involving 20 developers and researchers. The results of this evaluation demonstrated acceptance of the approach among software and mobile app developers. From the responses to our user evaluation RAPPT can be used as a starting point

¹⁰<https://github.com/ScottyB/rappt/blob/master/criteria.md>

to get to the first version of the code base up and running rapidly. Although we have not evaluated the time it took our users to develop the sample application as opposed to manually implementing the features in the code, we have received qualitative responses that confirm using RAPPT improves productivity. Future work will look at quantifying how much of a productivity boost RAPPT provides in comparison with implementing the feature from scratch and using other model driven approaches. Another avenue of future work would be to build up the DSL to include more concepts and to provide a constraint checking mechanism. Validating and verifying the visualisations prior to code generation will also be an interesting area of future work. Finally, addressing the issue of round trip engineering is another area that would improve the current work.

References

- [1] F. Bentley, E. Barrett, Building Mobile Experiences, The MIT Press, 2012.
- [2] S. Barnett, R. Vasa, J. Grundy, Bootstrapping mobile app development, in: Proceedings of the 2015 IEEE/ACM International Conference on Software Engineering (ICSE 2015), IEEE, 2015, pp. 305–306.
- [3] F. T. Balagtas-Fernandez, H. Hussmann, Model-driven development of mobile applications, in: Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on, IEEE, 2008, pp. 509–512.
- [4] J. Danado, F. Paternò, A prototype for eud in touch-based mobile devices, in: Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on, IEEE, 2012, pp. 83–86.
- [5] B. Athreya, F. Bahmani, A. Diede, C. Scaffidi, End-user programmers on the loose: A study of programming on the phone for the phone, in: Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on, IEEE, 2012, pp. 75–82.
- [6] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al., Scratch: programming for all, Communications of the ACM 52 (11) (2009) 60–67.

- [7] D. Wolber, App inventor and real-world motivation, in: Proceedings of the 42nd ACM technical symposium on Computer science education, ACM, 2011, pp. 601–606.
- [8] R. Francese, M. Risi, G. Tortora, Iconic languages: Towards end-user programming of mobile applications, *Journal of Visual Languages & Computing* 38 (2017) 1–8.
- [9] C. Rieger, Business apps with maml: a model-driven approach to process-oriented mobile app development, in: Proceedings of the Symposium on Applied Computing, ACM, 2017, pp. 1599–1606.
- [10] M. Naab, S. Braun, T. Lenhart, S. Hess, A. Eitel, D. Magin, R. Carbon, F. Kiefer, Why data needs more attention in architecture design - experiences from prototyping a large-scale mobile app ecosystem, in: Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, 2015, pp. 75–84. doi:10.1109/WICSA.2015.13.
- [11] A. Shye, B. Scholbrock, G. Memik, Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2009, pp. 168–178.
- [12] S. Barnett, R. Vasa, A. Tang, A conceptual model for architecting mobile applications, in: Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, 2015, pp. 105–114. doi:10.1109/WICSA.2015.28.
- [13] S. Barnett, Extracting technical domain knowledge to improve software architecture.
- [14] D. Steiner, C. Turlea, C. Culea, S. Selinger, Model-driven development of cloud-connected mobile applications using dsls with xtext, in: Computer Aided Systems Theory-EUROCAST 2013, Springer, 2013, pp. 409–416.
- [15] H. Heitkötter, T. A. Majchrzak, H. Kuchen, Cross-platform model-driven development of mobile applications with md 2, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, pp. 526–533.

- [16] O. Le Goaer, S. Waltham, Yet another dsl for cross-platforms mobile development, in: Proceedings of the First Workshop on the Globalization of Domain Specific Languages, GlobalDSL '13, ACM, New York, NY, USA, 2013, pp. 28–33. doi:10.1145/2489812.2489819. URL <http://doi.acm.org/10.1145/2489812.2489819>
- [17] H. Behrens, Mdsd for the iphone: developing a domain-specific language and ide tooling to produce real world applications for mobile devices, in: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, ACM, 2010, pp. 123–128.
- [18] I. Madari, L. Lengyel, T. Levendovszky, Modeling the user interface of mobile devices with dsls, in: Proc. of the Computational Intelligence and Informatics 8th International Symposium of Hungarian Researchers, 2007, pp. 583–589.
- [19] X.-S. Li, X.-P. Tao, W. Song, K. Dong, Aocml: A domain-specific language for model-driven development of activity-oriented context-aware applications, *Journal of Computer Science and Technology* 33 (5) (2018) 900–917.
- [20] G. Taentzer, S. Vaupel, Model-driven development of mobile applications: Towards context-aware apps of high quality., in: PNSE@ Petri Nets, 2016, pp. 17–29.
- [21] D. Bolchini, A. Faiola, The fusing of paper-in-screen: Reducing mobile prototyping artificiality to increase emotional experience, in: Design, User Experience, and Usability. Theory, Methods, Tools and Practice, Springer, 2011, pp. 548–556.
- [22] M. De Sá, L. Carriço, A mobile tool for in-situ prototyping, in: Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services, ACM, 2009, p. 20.
- [23] A. P. Jørgensen, M. Collard, C. Koch, Prototyping iphone apps: realistic experiences on the device, in: Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, ACM, 2010, pp. 687–690.

- [24] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas, T. Kosar, On the use of a domain-specific modeling language in the development of multiagent systems, *Engineering Applications of Artificial Intelligence* 28 (2014) 111–141.
- [25] A. A. Shah, A. A. Kerzhner, D. Schaefer, C. J. Paredis, Multi-view modeling to support embedded systems engineering in sysml, in: *Graph transformations and model-driven engineering*, Springer, 2010, pp. 580–601.
- [26] H. Zhao, L. Apvrille, F. Mallet, Multi-view design for cyber-physical systems, in: *PhD Symposium at 13th International Conference on ICT in Education, Research, and Industrial Applications*, 2017, pp. 22–28.
- [27] F. A. Kraemer, Engineering android applications based on uml activities, in: *Model Driven Engineering Languages and Systems*, Springer, 2011, pp. 183–197.
- [28] J. Sauro, J. R. Lewis, When designing usability questionnaires, does it hurt to be positive?, in: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11*, ACM, 2011, pp. 2215–2224.

4.2 Specifying Model Transformations by Direct Manipulation using Concrete Visual Notations and Interactive Recommendations

Avazpour, I., Grundy, J.C., Grunske, L. Specifying Model Transformations by Direct Manipulation using Concrete Visual Notations and Interactive Recommendations, *Journal of Visual Languages and Computing*, Volume 28, June 2015, Elsevier, pp 195–211.

DOI: [10.1016/j.jvlc.2015.02.005](https://doi.org/10.1016/j.jvlc.2015.02.005)

Abstract: Model transformations are a crucial part of Model-Driven Engineering (MDE) technologies but are usually hard to specify and maintain for many engineers. Most current approaches use meta-model-driven transformation specification via textual scripting languages. These are often hard to specify, understand and maintain. We present a novel approach that instead allows domain experts to discover and specify transformation correspondences using concrete visualizations of example source and target models. From these example model correspondences, complex model transformation implementations are automatically generated. We also introduce a recommender system that helps domain experts and novice users find possible correspondences between large source and target model visualization elements. Correspondences are then specified by directly interacting with suggested recommendations or drag and drop of visual notational elements of source and target visualizations. We have implemented this approach in our prototype tool-set, CONVERt, and applied it to a variety of model transformation examples. Our evaluation of this approach includes a detailed user study of our tool and a quantitative analysis of the recommender system.

My contribution: Developed initial ideas for this research, co-designed approach, co-supervised PhD student, wrote substantial parts of paper, investigator for funding for this project from ARC

Specifying Model Transformations by Direct Manipulation using Concrete Visual Notations and Interactive Recommendations

Iman Avazpour^a, John Grundy^a, Lars Grunske^b

^a*Centre for Computing and Engineering Software and Systems (SUCCESS), Swinburne University of Technology, Hawthorn 3122, VIC, Australia*

^b*Institute of Software Technology, Universität Stuttgart, Universitätsstraße 38, D-70569 Stuttgart, Germany*

Abstract

Model transformations are a crucial part of Model-Driven Engineering (MDE) technologies but are usually hard to specify and maintain for many engineers. Most current approaches use meta-model-driven transformation specification via textual scripting languages. These are often hard to specify, understand and maintain. We present a novel approach that instead allows domain experts to discover and specify transformation correspondences using concrete visualizations of example source and target models. From these example model correspondences, complex model transformation implementations are automatically generated. We also introduce a recommender system that helps domain experts and novice users find possible correspondences between large source and target model visualization elements. Correspondences are then specified by directly interacting with suggested recommendations or drag and drop of visual notational elements of source and target visualizations. We have implemented this approach in our prototype tool-set, CONVERt, and applied it to a variety of model transformation examples. Our evaluation of this approach includes a detailed user study of our tool and a quantitative analysis of the recommender system.

Keywords: Model Driven Engineering, Model Transformation, Visual Notation, Recommender System, Concrete visualizations

Email addresses: iavazpour@swin.edu.au (Iman Avazpour), jgrundy@swin.edu.au (John Grundy), lars.grunske@informatik.uni-stuttgart.de (Lars Grunske)

1. Introduction

Model transformation plays a significant role in the realization of Model Driven Engineering (MDE). Current MDE approaches require specifying correspondences between source and target models using textual scripting languages and the abstract representations of meta-modeling languages. Although these abstractions provide better generalization, and hence code reduction, they introduce difficulties for many potential transformation specification users. This is because in order to effectively use them, users have to possess in-depth knowledge of transformation languages and meta-modeling language syntax. These are often very far removed from the actual concrete model syntax for the target domain. Moreover, taking into account large models being used in today's software systems, many transformation specifications are very complex and challenging to specify and then maintain, even for experienced transformation script and meta-model users [1, 2, 3]. Although some approaches have been developed to mitigate these problems, such as by using visual abstractions [4, 5], by-example transformations [3, 6, 7], graph transformations [8, 9, 10, 11], automatic inference of bi-directional transformations [12, 13], and automated assistance for mapping correspondence deduction [14], none of these fully address the problems nor do so in an integrated, visual, human-centric and highly extensible way.

We introduce a new approach that helps to better incorporate user's domain knowledge by providing them with familiar concrete model visualizations for use during model visualization and transformation generation. This approach follows the three principles of *direct manipulation* [15], i.e. (1) it provides support for generating concrete visualizations of example source and target models; (2) these visualizations allow user interaction in the form of drag and drop of their concrete visual notation elements; and (3) interactions are automatically translated into transformation code and hence direct coding in complex transformation scripting languages is avoided. In addition, to better aid users in finding correspondences in large model visualizations, an automatic recommender system is introduced that provides suggestions for possible correspondences between source and target model elements. Complex model transformation code is automatically generated from the user's interaction with concrete visual notations and suggested recommendations.

This paper is organized as follows: Section 2 gives a motivating example,

our key research questions and the requirements being addressed by the research reported in this paper. Section 3 briefly discusses key related work. Section 4 outlines our approach to model transformation generation followed by a usage example in section 5. Section 6 describes the architecture and implementation of our approach in CONcrete Visual assistEd Transformation (CONVERt) framework. Section 7 describes our evaluation and user-study setup and is followed by a discussion in section 8. Finally section 9 concludes the paper with a summary.

2. Motivation

Assume Tom, a software developer, is working in an MDE-using team and has received a system analysis report for an application. Being an expert in UML diagram interpretation and a Java coder, he is familiar with concrete syntax of the diagrams and Java code. He is interested in transforming specific parts of UML diagrams provided by the analysis directly to his programming code, to increase team productivity, code quality and to ease software evolution. For example, he wants to create a model to code translator in order to transform specific features and parts in the analysis diagrams to specific Java code templates. For Tom, as an expert in the domain, corresponding elements in the UML diagram and in his Java code are obvious. He can clearly spot and relate classes, methods, and even statement snippets in both program code and class diagram. For example, he can easily relate an attribute in a class diagram to a property in Java code and their fine-grained elements (i.e. types, names and access identifiers). Some such model element correspondences are depicted by Figure 1., using concrete visualizations of the UML model (a class diagram) and code model (Java textual syntax).

As another example, consider Jerry, an urban planner, who is preparing a report on traffic congestion in part of a city. He is used to viewing volumes of vehicles crossing intersections on screen using a geo-spatial visualization. An Example of this visualization is shown on the left side of Figure 2. Here, the volume of vehicles are represented on a map using bubbles. In his report, he would like to reflect the volume of vehicles passing set of intersections in a particular time instance by a pie chart. Being an expert in this domain, he has a solid understanding of this map-based visualization and pie charts and therefore their corresponding relationships are obvious to him. He would like to relate the number reflected to each bubble to a pie piece in a pie chart and generate new visualizations for his report as shown by Figure 2.

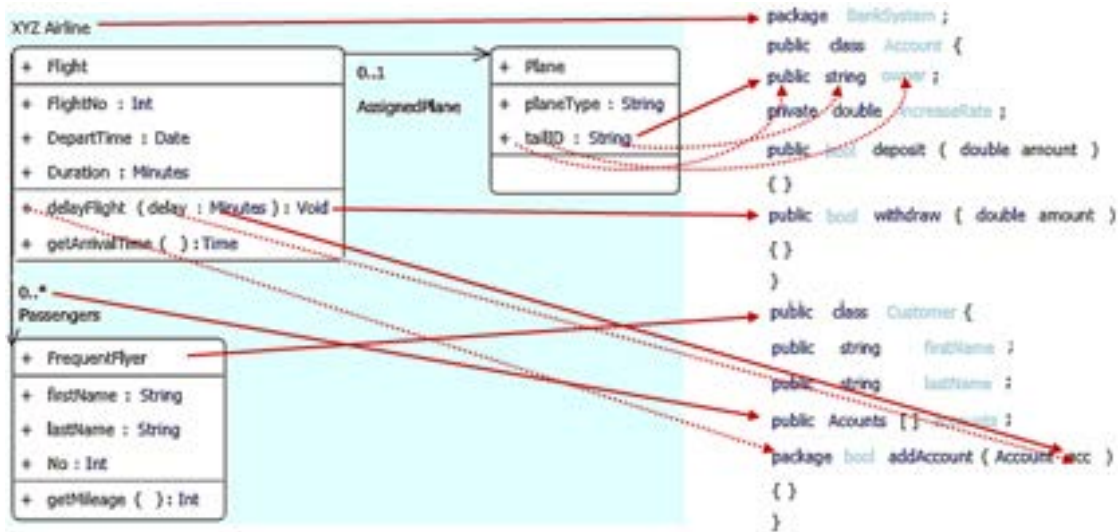


Figure 1: Example of correspondence relations between a Class Diagram and Java code. Dashed arrows show more fine-grained correspondences.



Figure 2: Example of correspondence relations between geo-located bubbles and pie pieces in a pie chart.

Given that Tom and Jerry may not have experience or knowledge of transformation languages, meta-modeling, and data processing, specifying correspondences is better understood by them using one or more example visualizations (e.g. class diagrams and corresponding code examples, as shown in Figure 1). However, using current approaches to create such transformations they have to work with the complex syntax of model abstractions (e.g. UML and Java meta-models, etc.) and the low-level textual syntax and semantics of transformation languages, such as XSLT, ATLAS Transformation Language (ATL)¹ and Query/View/Transformation (QVT). Additionally, with these approaches, maintaining the transformations if the source or target models need to change is time-consuming and error-prone. While

¹<http://www.eclipse.org/atl/atlTransformations>

a transformation expert may be able to do this, novices can easily make mistakes or find difficulties. If the source and target models are large, mapping is even more difficult and time-consuming, even with good tool support, such as Altova². Such issues make current transformation specifications hard to perform, understand and evolve (e.g. adding or changing new transformation rules). This problem exists in many other transformation domains, for example, model to model transformations (e.g. UML to RDBMS), model refactoring, graph based transformations, and most information visualization domains.

In earlier work we developed prototype solutions to some of these problems, but for specific domains and addressing only limited parts of the problem domain. For example, the EDI message mapper using abstract EDI structure [16], the form-based mapper using a concrete business form metaphor [17], and mapping agents for large meta-model mapping problems [14]. Key research questions we wanted to answer in the work described in this paper include:

1. Can we generalize the use of such concrete source-to-target mapping metaphors to a wide range of model transformation problems?
2. Can we generate efficient and effective complex model transformation scripting code from these visual by-example specifications?
3. Can we provide effective guidance to users with visual representations and recommendations for large model mapping problems?

To answer these questions, we seek to meet the following key requirements in our approach and associated tool-set: (i) provide users with familiar concrete visualizations of source and target models in order to leverage their domain knowledge in transformation specification; (ii) allow users to specify complex model element mappings between concrete visual notational elements using interactive drag-and-drop and reusable, spreadsheet-like mapping formula; (iii) help users find, explore and decide possible model correspondences by providing automated recommendations using an interactive recommender system; (iv) allow users to cut corners in specification of transformation correspondences by choosing among suggestions; (v) automatically create high-level abstractions for transformation generation from the concrete visualizations; and (vi) generate reusable model transformation implementations from these visually specified, by-example model mappings.

²<http://www.altova.com/>

This paper presents our approach for supporting these tasks in a proof-of-concept visualization and model transformation specification and generation tool. In the following sections, we show the practicality of our transformation approach for generating transformers for a variety of domains.

3. Related Work

Complexity of model transformation specification is due to both the difficulty inherent in specifying large, complex inter-model correspondences using the complex syntax of model transformation languages [17, 18], and the use of meta-models [19]. Most model transformation approaches rely on knowledge of multiple, often large and complicated meta-models, along with expert knowledge of transformation scripting languages and tools (e.g. Eclipse Modelling Framework project [20]). While these provide powerful platforms, they are also time-consuming to maintain, hard to understand and error-prone to write [2, 3]. Multiple approaches have been proposed to address the complexity of transformation specification by eliminating the need for learning transformation languages and dealing with meta-models. These approaches can be grouped into Model Transformation By Example (MTBE), Model Transformation By Demonstration (MTBD) and model and meta-model matching.

The key principle with the MTBE approach is to derive high level model transformation rules from an initial prototypical set of interrelated source and target models. This concept was first used by Varro et al. incorporating graph transformations [6]. The idea is to provide multiple source and target model pairs, and ask a user (domain expert) to specify source and target model element correspondences. The system then uses these correspondences to derive transformation rules [1, 3, 6, 7]. Model matching approaches are very similar to MTBE, i.e. they try to find possible correspondences between source and target models. These techniques try to find an alignment for relating two or more models. This alignment can then be used to semi-automatically generate transformations between two models [21, 22, 23]. These generated transformations can then be adopted and validated by an expert as a set of transformation rules. Both MTBE and model matching approaches usually require multiple source and target model examples to exist, ideally representing the same underlying data, to produce accurate transformation rules. Moreover, they often require users to modify derived rules

to be executable and to match their intention which are often harder than writing the rules from scratch [18].

MTBD approaches on the other hand are based on an expert performing transformation tasks and recording of the process steps by a recorder. Using the recorded history, the system generalizes the tasks to form abstract transformations [2, 24]. MTBD approaches are more suited for transformation of models conforming to the same meta-model and as a result, are more difficult to use for different model transformation applications [1].

Kappel et al.[1] and Li et al. [17] have separately argued that to specify model transformations, an approach that involves concrete notations is required. Kandel et al. mentioned that analysts could benefit from interactive tools that simplify the specification of data transformations [25]. They asked whether transformations can be communicated unambiguously via simple interactive gestures over visualized data or if relevant operations can be inferred and suggested [25]. Multiple approaches have been used in that direction, leveraging concrete model notations for the specification and development of transformation rules and data mappings [26, 27]. However, they are usually hard coded for specific problem domains or use fixed visualizations [17]. In addition, they mostly require users to specify correspondences on meta-models rather than concrete visualizations [10, 28, 29, 16, 4]. This is also true for graph based transformations, where source and target models are represented using abstract graphs [8, 9, 10, 11]. There has been works on using graphical representations on graph-based transformation languages. For example a graphical representation of model transformations for Triple Graph Grammar (TGG) was provided by Grunske et al. [30]. Concrete syntax-based Graph Transformation (CGT) was introduced by Gronomo et al. [31]. It was suggested that due to use of graphical concrete syntax, CGT is more concise and requires considerably less effort from the modeler than , ATL and Attributed Graph Grammar (AGG) which use textual abstract syntax [31]. CGT uses a default concrete syntax similar to Business Process Modeling (BPM) and therefore the syntax is familiar for the modeler's domain knowledge. However, this concrete syntax does not have a flexibility of adapting to arbitrary visualizations.

In contrast, the approach presented in this paper uses concrete visualizations and can be adopted for variety of domains. Also, it uses model matching concepts and provides a recommender system that specifically focuses on concrete visual model representations to guide users in specifying their transformation rules. We demonstrate our approach using MDE case study

of transforming class diagrams to Java code snippets. This case study and automatic model transformation and code generation in general has been an active field of research in software engineering for many years and is a necessity for MDE [32, 33, 34, 35, 36, 37]. These and other model transformations in MDE has been practiced for many years with graphical or textual languages (e.g. Henshin [33] or ATL). However, the biggest barrier for developing such code generators and transformers lies with the complexity of the model transformations and the capability of users to correctly specify the transformation rules. Previous approaches to code generation used template generators [14, 17], patterns [38], and textual and/or visual meta-models [16, 39]. We hope that by using concrete visualizations, we can provide facilities to model transformation users to better integrate their domain knowledge in transformation rule generation. Our approach generalizes to many other domains and we have applied it to several diverse data migration, data aggregation and complex information visualization problems.

4. Our CONVERt approach

Our approach to model transformation generation in CONVERt relies on example concrete visualizations of input source and target models. These specifically generated visualizations enable use of drag and drop of notations to perform model transformations and specify correspondences between source and target model visualizations. Consecutively, this concrete, by-example approach for model transformation has three key steps: (1) The user - the domain expert - provides source and target model examples and specifies (or reuses) a concrete visualization for each of the provided examples (a model to visualization transformation). These source and target model visualizations may be very different from one another e.g. a UML class diagram visualization and Java code visualization, or a map visualization and a chart visualization. (2) Using these example model visualizations, the user interactively specifies mapping correspondences between source and target visualization elements (visualization to visualization transformation). (3) We generate reusable model transformation script code from the specifications. This reusable script can be applied to any source model conforming to the example source model(s) used to specify the transformations, to produce a target model. In the following we describe each step of our approach in more detail. Then we provide a usage example of our approach being used to specify and generate model transformations.

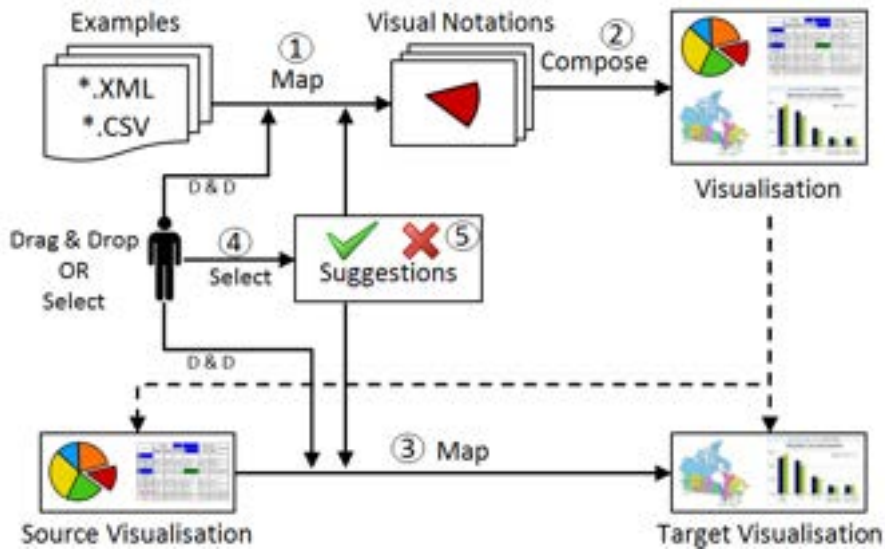


Figure 3: CONVERt’s transformation approach.

CONVERt’s approach is outlined in Figure 3. In step one, a user provides at least one example of source and target models. Our proof of concept implementation of the approach uses XML or Comma Separated Value (CSV) files as input examples and can be further expanded to allow other types of input. Using these example inputs a concrete visualization needs to be specified and generated for both source and target examples (or reused, if concrete visualizations have previously been specified for other example models of these types). To do this, users map elements of the source and target examples to available visual notations provided by our tool framework and specify their correspondences (see 1 in Figure 3).

Because models to transform are usually very large, a recommender system, a “Suggester”, analyses the provided examples and available notations and generates a list of possible transformation correspondences to be presented to the user (see 5 in Figure 3). Our suggester uses multiple similarity heuristics, including model element name, value, structure and neighborhood similarities. It aggregates the results returned by each heuristic to generate a model element correspondence suggestion list. To specify input example to visual notation mapping correspondences, users can either drag and drop elements of input examples on visual notation elements or select from recommended correspondences (see 4 in Figure 3). When specified, users compose these visual notations to create a complete concrete visualization for models of each of the source and target types (2 in Figure 3). These mappings result in generation of model transformation code that transforms model elements

to concrete visualizations. Note that this visualization step can be specified separately by other users or be reused from previous visualizations.

Using the generated visualizations, users specify transformations between elements in the source model and elements in the target model using the concrete visualizations (see 3 in Figure 3). These visualization to visualization mappings are similarly done by dragging and dropping elements of source visualizations on elements of target visualizations or selecting from provided suggestions. For example, a pie piece in a pie chart can be dragged and dropped on a bar in a bar chart to specify a mapping correspondence; Hence the process of model transformation can use user's domain knowledge.

Correspondences between model elements include 1 to 1, 1 to many, many to 1 and many to many element correspondences. Often these transformations are quite complex. To achieve these, a variety of model transformation functions, such as collection summation, merging, subtraction, textual parsing and conditional mapping, are provided to users. In keeping with our visual, by-example transformation specification metaphor, these are also applied to example concrete visual element(s) and are composed together visually. These enable tool users to specify potentially very complex model transformation rules. If required, users can also define new functions using provided templates.

The visually, by-example defined model correspondences are then translated into low-level model transformation rules, currently implemented in XSLT. Once all required transformation rules are defined, the system generates a full source model to target model transformation script. This transformation code can be applied to any source model examples conforming to the meta-model of the example(s) used in the specification to produce a target model.

5. Use Case

Assume Tom, a software engineer, intends to create an automatic code generator to transform specific parts of a UML class diagram model to Java code. While various IDEs and template generators support generic code generation, Tom may want to specify particular pattern implementations be used, particular code snippets be used, particular code formatting, commenting and layout, use of particular APIs be used in particular ways within the generated code base, and may want particular coding approaches be implemented. Let us further assume that he has example XML representations of

both class diagrams and target Java code examples. Tom will provide these examples to CONVERt. Note that it is not required for him to possess multiple examples; instead, one comprehensive example would suffice to support the transformation generation procedure. However, having more examples will help the suggestion mechanism calculate more accurate recommendations and will help CONVERt generalize from the examples to a more complete generated translator. Specifying using several, smaller examples can also be easier for users to work with than one large example source and target model.

If meta-models are available for these example models, then CONVERt can use these for validating its input and generated target models, constraining rules and improving transformation code generation. Otherwise, CONVERt will automatically reverse engineer a meta-model from the provided example models. This meta-model is used in transformation rule templates and the suggester system.

5.1. Specifying Concrete Model visualizations

The model visualization procedure (Step 1) involves creating a visual notation for each distinct part of input model once and composing them together to form a complete visualization. For example, to visualize an example Java code XML, Tom needs to define a visual notation for Java package, Classes, attributes, methods and method parameters. To do that, he has to specify correspondence links between elements of the notation and his input. Visual notations are provided by framework users or designers and are available in framework's notation repository for reuse. Available visual designs can be imported in the framework and registered as notations by providing a notation to data mapping (See [40] for more information on CONVERt's support for specifying and generating reusable visual notations).

For example, using CONVERt to generate a visualization for a Java class, he has to drop a previously defined or reused *Java class notation* onto the CONVERt designer canvas (see (1) in Figure 4(a)). He then drags and drops the *class element* of the source example model on the notation as shown by solid black arrow in Figure 4(a). This interaction will trigger the creation of a transformation rule for transforming that portion of the source model (the *class* element in the source XML document) to the host notation's model. Each notation may have internal elements which are accessible through a pop-up window. For example, our class notation here has an access identifier, a name, a placeholder for properties and a place holder for methods as its



(a) Defining Java class notation.

(b) Defining UML class notation.

Figure 4: Mapping example elements to notational elements to define visual notations for a) Java class and b) UML class. Arrows depict drag and drop.

model (see (2) in Figure 4(a)). These placeholders specify where other visual notation elements are going to be included.

Tom defines correspondences between the source model element(s) and target concrete visualization element(s) by drag and dropping elements or choosing from automatically-generated “suggestions” (see (3) in Figure 4(a)). In this example, Tom drags and drops the input class’s name and access to name and identifier of the notation (dashed arrows in Figure 4(a)). These correspondences will be included in the transformation template that has been triggered. Attributes and methods are defined by other model to visual notation mappings, and are specified in the same way. Thus we do not need to map them to these place holders at this stage. Once done, he will save the notation and continue creating other required notations. The same procedure is then followed for other notational elements and their visualizations. For example, a UML class diagram’s Class notation is shown in Figure 4(b).

Complex 1-to-many, many-to-1 and many-to-many element correspondences can be specified. To facilitate this and other complex mapping tasks, a range of “mapping functions” are available in CONVERt. For example, if Tom wanted to alter the name of the Java classes by appending a “_Class” to their name, he could use a string merging function (marked by *A* in Figure 5(a)). These mapping functions are used in a similar way to the notation elements, i.e. Tom drops the required function on the designer canvas, and links desired input elements to internal elements of the function (i.e. function’s input arguments) by drag and drop, and drags the output of the function to

his desired element in the notation (see arrows in Figure 5). This forms a data-driven functional visual language transforming the source model data to the target visualization elements in potentially very complex ways.

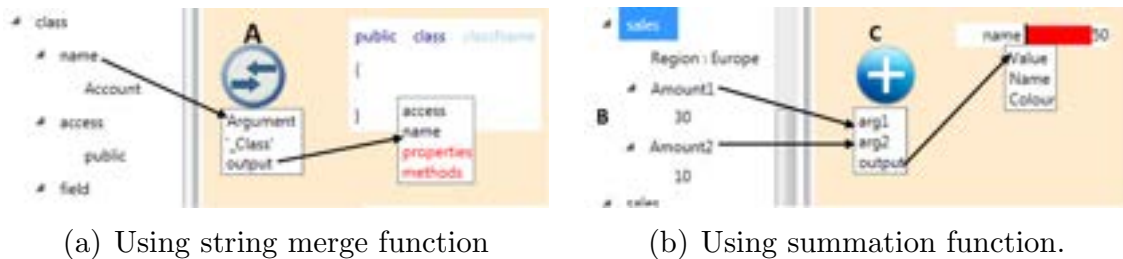


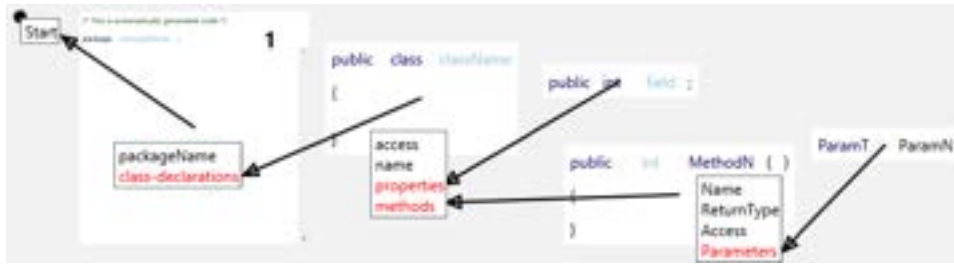
Figure 5: Using functions for input to notational element mapping. a) string merge, and b) summation function. Arrows depict drag and drop.

To get a sense of range of supported model transformations and different domains, assume analyst Carrie is generating a bar chart visualization for her sales report in a business analysis visualization domain. Each bar in this bar chart is to represent a record of annual sales. Further assume the report being handed to her includes six monthly sales amounts instead of annual sales (marked by *B* in Figure 5(b)). She can use a summation function (*C* in Figure 5(b)) to add those two amounts and map the result to Bar’s value.

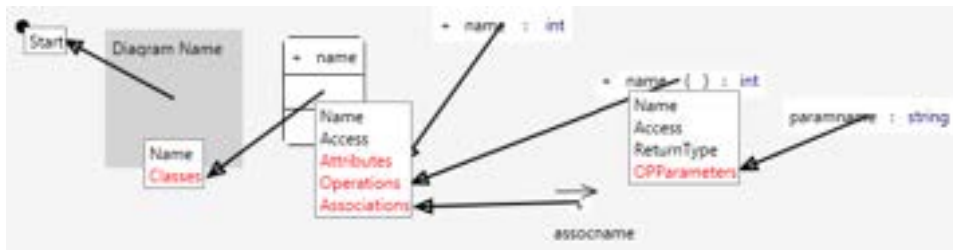
The defined notations represent a model element-to-visual notation transformation rule. To have a complete transformation script, the prepared collection of transformation rules in notations need to be scheduled for processing on source models according to their call sequence. Usually this is achieved by asking users to write code for this script, similar to procedural programming, and by providing meta-models. In our approach the assumption is that there is no user provided meta-model available and the user is not willing to write complex transformation code. Therefore, by using composition of notations our approach generates call sequencing of the embedded transformation rules.

To compose the defined notations in our example, Tom links all notations he has created according to their specific place holders as depicted by Figure 6(a). By linking a notation to a placeholder element of another, the host notation knows the transformation rule embedded in the notation being dragged should be called at this placeholder. This is in order to affect the embedded model element-to-visual notation mapping. This linking results in the scheduling of model element-to-visual notation transformation rules and thereby creates the model to visualization transformation script.

A *Start* element in Figure 6 defines where the transformation specifica-



(a) Java code notation composition.



(b) Class diagram notation composition.

Figure 6: Notation composition to generate visualizations for a) Java code and b) class diagram. Arrows are provided by the framework for better tracking of the composition.

tion will start and thus tells the transformation scheduler to start generating transformation code from the transformation rule embedded in the notation linked to this start element. For example in Figure 6(a), the transformation rule in package notation (marked by 1) is the first rule to be called to transform a Java package model element to a package notation. It then calls the class transformation rule, and the scheduling continues accordingly for other linked notations.

Each notation composition results in the automatic generation of a transformation specification, currently implemented as XSLT transformation script that generates Windows Presentation Foundation (WPF) visual elements. For example, by using the compositions specified in Figure 6(b), a complete XSLT script to generate concrete visualizations of UML class models will be generated for rendering class model examples to visualizations of the form in Figure 1. Note that this generated XSLT transformation script can be reused and applied to all Java code and class diagram model files conforming to the examples used to specify the visualizations, to provide an automatic concrete visual notation renderer. These generated concrete class visualizations are implemented as WPF elements and allow interaction with their composing notations. The individual elements of a concrete visualization can thus be dragged, dropped on other elements, and right clicking on them reveals their internal elements.

5.2. Specifying Model Mappings using Concrete Visualizations

Once visualizations of both source and target are available, Tom can drag and drop elements of these visualizations to create transformation rules between these visualizations. An alternative to this approach is to select from provided suggestions (see *B* in Figure 7). Figure 7 shows an example of creating a transformation rule for a UML attribute to a field in Java code. To create this rule, Tom needs to drag a UML attribute to a Java field, as depicted by solid black arrow, and match their internal elements, as shown by dashed black arrows (or select them in suggested recommendations). Note that the two visualizations do not need to represent same data, as in Figure 7 where the class diagram represents an organization system but the Java code is representation of a Bank system package. Although consistency checking was not our concern with this prototype implementation, selective checks can be provided in each visualization. For example, the package name of the Java code knows that its name should not consist of white spaces, or Java attributes use default multiplicity of *1* when not specified and when blank is provided it is assume to be *n*. These checks can be provided depending on the application during notation design. An alternative is to use functions and conditions when specifying the transformations.

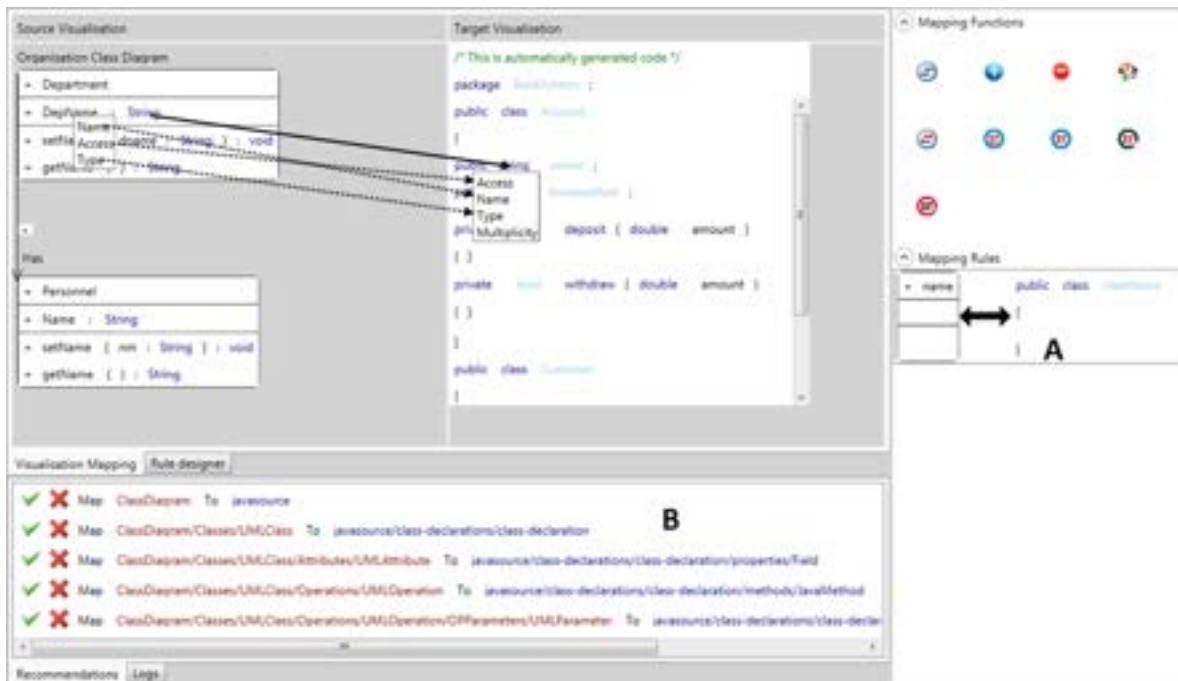


Figure 7: Mapping A UML attribute to Java property. Arrows depict drag and drop directions.

Given that transformation rules are defined using concrete notations, each

rule is also represented by the source and target notations it is representing. This provides better visual representation of transformation rules. For instance in Tom's example above, the visual representation of the rule transforming a UML class to a Java class is marked by *A* in Figure 7.

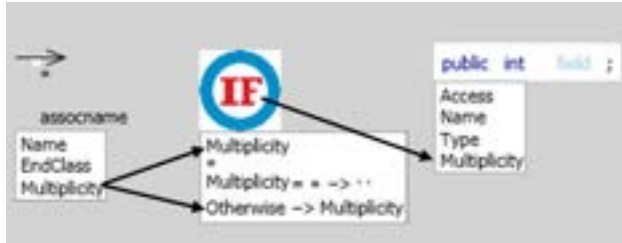
Similar to step two of specifying a concrete visualization for a model, mapping functions are available to create more complex transformation rules between the concrete visual model mappings. For example, when mapping associations to a Java field property, an association might have multiplicity defined by "*", whereas a Java field property might have either a number or void as its multiplicity. To specify such a correspondence, Tom can use a condition mapping function.

By dragging a UML attribute to a Java field (or any notational element to another) their default notations will be shown on a different canvas to better provide space for using functions. In this example, Tom can navigate to that canvas and specify conditions as depicted by Figure 8(a). The condition function in the figure tests whether *Multiplicity* of the association is equal to '*'. If so, it passes a blank character as output; otherwise it copies the *Multiplicity* provided by the association to the output. Since conditions do not have specific output, (unlike arithmetic and processing functions), instead of dragging the output he drags the condition itself on the element of the target (in this case Java field's multiplicity) as depicted by arrows in Figure 8(a). He can continue specifying other correspondences (Associations *Name* to Java field *Name* and *EndClass* to *Type*) here or on the actual visualizations. The transformation code generated from these interactions is shown in Figure 8(b).

5.3. Recommending correspondences

In our experience, real-world source and target models are often very large [14]. To assist the user, model correspondence recommendations are provided by a group of *Suggesters*, or correspondence recommenders, which analyze source and target models according to a variety of value and structural similarity heuristics. Since analyzing the whole input models and visualizations was costly for our automatic correspondence recommender, our suggester system uses the abstract graph lattices (used in reverse engineering meta-models) as input to calculate similarities.

Three types of similarity heuristics are used as correspondence recommenders of the suggester system. (1) Static similarity recommenders check name tag and type of elements in source and target elements. (2) Structural



(a) Mapping UML Associations to Java field

```

<xsl:template match="UMLAssociation"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <Field>
    <Access>public</Access><!--default-->
    <Name>
      <xsl:apply-templates select="Name" />
    </Name>
    <Type>
      <xsl:apply-templates select="EndClass" />
    </Type>
    <Multiplicity>
      <xsl:choose>
        <xsl:when test="Multiplicity='*' '>' '<!--default-->
        <xsl:otherwise>
          <xsl:value-of select="Multiplicity"/>
        </xsl:otherwise>
      </xsl:choose>
    </Multiplicity>
  </Field>
</xsl:template>

```

(b) Generated XSLT script

Figure 8: Using conditions in visual notation mapping, a) a condition to map UML association to Java field, arrows depict drag and drop. b) The resulting XSLT script.

similarity recommenders consider source and target as graphs and check similarities based on structure of this graph that elements reside in, i.e. checking the inbound or outbound nodes of each element, and checking the neighborhood of each source and target element. (3) Propagated similarity (like structure similarity recommenders) considers input source and target models as graphs and calculates similarity of elements according to recursive analysis of their neighboring elements. With this similarity measure, similarity of two nodes in a graph is defined by similarity of their neighborhood topology. As a result, using propagated similarity, two nodes are similar if their neighbors are similar and the neighbors of their neighbors are similar and so on. We have adopted IsoRank as our measure for propagated similarity recommender [41].

Figure 9 shows sample list of recommendations produced by our suggester for the motivating example of UML class diagram to Java code visualization. In this figure for example, a name similarity recommender has assigned high scores to UML class Name to name of a Java class and UML class's Access to Java class's access as they represent similar name tags. UMLClass and class-declaration have high similarity score according to IsoRank similarity as their neighbors (children and parents) are similar (e.g. UMLClass children include Name and Access, Java class declaration also has name and access as it's children and so on). As a result they have been returned in the final list of recommendations. Note that other recommenders might have also contributed to these recommendations. For example, both Name elements

on either side have values that are *string* and the type similarity recommender will also return these pairs as high score correspondence candidates.



Figure 9: Sample of resulting suggested correspondences between UML class diagram and Java code visualization.

The scores calculated from the similarity recommenders of each group are returned as a normalized similarity matrix. Similarity matrices are sent to the Suggester system and a final similarity score is calculated based on the confidence weights assigned to each recommender in the suggester’s recommendation ensemble. Similarity scores returned by suggesters are multiplied by their confidence weights and the resulted scores are summed up in a final similarity matrix that is the basis for calculation of recommended correspondences. The suggester system selects from the returned suggestions and prepares a recommendation list similar to Figure 9. Once all recommendations are available, our ensemble configuration filters the recommendation list by the stable marriage algorithm [42]. This will result in a selection of recommendations that possess the highest overall recommendation score per pair. The stable marriage algorithm can be configured to return arbitrary number of results per pair. By default this value is set to one. If users accept or reject any of the recommended correspondences a feedback analyzer updates the confidence weights associated with the suggesters and thus improves the learning mechanism.

The suggester system can be configured to use one, all or a selection of these different suggesters by provided option settings. For example, users can configure the suggester system to ignore a particular recommender by setting its usage flag to false. Also, it is possible to alter weights manually (or as part of an optimization mechanism) by setting each recommender’s confidence weights and disabling the learning mechanism (automatic alteration of confidence weights).

To provide a more interactive and hence useful representation of recommendations, the suggester system incorporates a filtering mechanism similar to the *guide* and *filter* approach proposed by Hernández del Olmo et al. [43].

In their proposal, a *guide* provides answer to when and how each recommendation must be shown to the user, while the *filter* must answer which of the items are useful/interesting candidates to become recommended items.



Figure 10: Adaption of *guide* and *filter* for interaction with suggested correspondences. Selecting (accepting) a UML class to Java class correspondence (1) updates the recommendation list to show possible internal element correspondences (2). Rejecting the correspondence will not update the list but will result in updating the recommender weights.

In our adaptation of that proposal, the results of final similarity matrix are *filtered* by the stable marriage and sent to the *guide* system for representation. Some correspondences will result in transformation rules, and others will define internal rule correspondences. The *guide* system chooses among recommendations according to the task that the user is about to perform, e.g. when user provides source and target visualization to perform mappings, the *guide* system first represents the recommendations that will result in transformation rule templates. That is because a rule between two notations must be defined first, and its internal rule correspondences are to be defined later. Therefore, when users define a rule correspondence by drag and drop or selecting from the suggesters, the system accordingly updates the list of suggested correspondences to provide suggestions related to that rule and hence better guide users with targeted recommendations. For example, if Tom defines a UML class to a Java class rule by accepting its recommended suggestion (as in Figure 10 A) or alternatively by drag and drop of their visual notations, the suggestion list will be updated to demonstrate how internal elements of classes (like name, access, attributes, etc.) can be linked (See B in Figure 10). This intelligent assistance was incorporated after our user study indicated the need for more interactive and targeted visualization of recommended correspondences.

5.4. Generating Model Transformation Scripts

Once the required rules for transforming all parts of source and target visualizations are defined, a transformation code generator generates XSLT scripts for each transformation rule in the form of an XSLT template similar to the transformation script of Figure 8(b). As stated before, it is possible to generate scripts for other transformation languages with some modifications in the transformation code generator. These modifications would specify how a correspondence from an element a in source to an element b in target should be written for that specific transformation language. In current configuration, depending on what the correspondences specify (an element to element mapping, a notation to notation mapping, etc.) these correspondences are translated to XSLT value fetches (value-of and select statement) or template snippets.

```
/* This is automatically generated code */
package OrganisationClassDiagram ;
public class Department
{
public String DepName ;
public Personnel [] Has ;
public void setName (String dname )
{ }
public String getName ( )
{ }
}
public class Personnel
r
```

Figure 11: Sample resulting Java visualization.

Although the model to visual notation transformation rules of the visualization step had to be explicitly scheduled, in visualization to visualization transformation generation the rules are declaratively executed. This is due to the fact that a visualization example of both source and target are available and their meta-model can be reverse engineered. Therefore, it is possible to decide the starting rule for the transformation script. Other transformation rules following the starting rule are then declaratively called. As a result, once required transformation rules are defined, the system will generate the

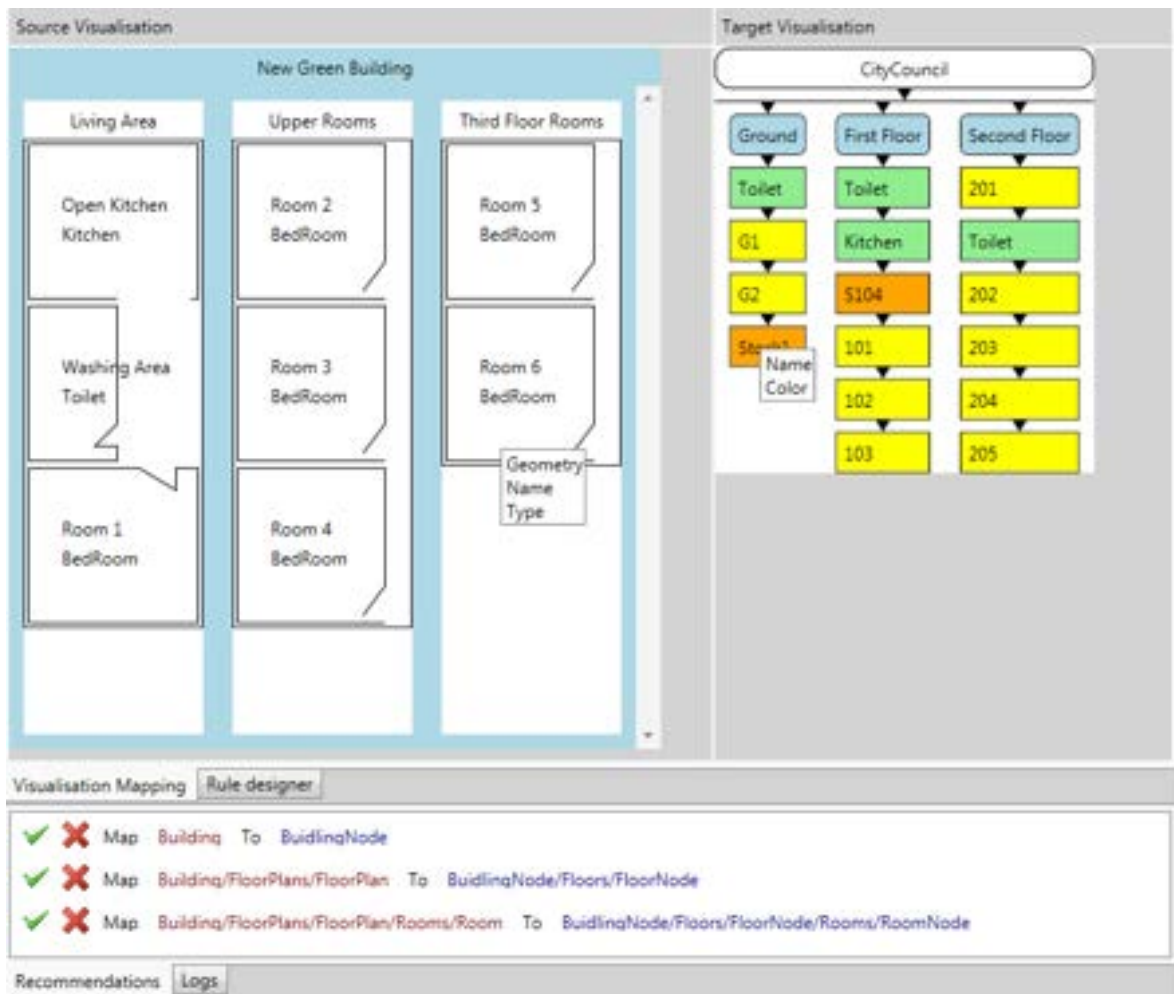


Figure 12: Using CONVERt to transform CAD drawings to tree-based layout.

transformation script automatically. Applying this script to source visualizations will transform the data represented by source to visualization of the target. For example, in Tom's example, applying the full transformation script on the source of Figure 7 will result in the visualization of Figure 11. Note that Tom can very easily modify the Java example visualization and mappings to: rearrange attributes and methods; generate differently-named classes, methods and/or attributes; to reformat their concrete appearance; to add particular design pattern, API usage, code formatting, layout, and partial code templates; could apply filtering to the source UML model mappings to only generate code for selected portions of the UML model; and so on.

5.5. A CAD Model Transformation Example

To illustrate the range of model transformation domains to which our CONVERt approach can be applied to, consider Computer Aided Design (CAD) applications that need to exchange complex models [44]. Consider the scenario where architect Carrie might want to create an organization’s building structure chart based on the design she had created earlier. Assume that visualization transformations for both models have been provided beforehand (they can be created using the same approach as in the previous example), where the design model is visualized with a 2D building layout and the structure chart model via its diagrammatic representation. Carrie can specify a transformation between elements of her source design to elements of the target structure chart. As she is an architect and not a software engineer, and the fact that CAD designs can become very large and complex, she can view both visualizations side by side and get help from suggested correspondences.

Figure 12 shows an example of mapping parts of a detailed building design to a detailed structure chart. Carrie can specify elements of the chart structure to be created based on elements in her design. For example, she can drag and drop a room on a corresponding room node in the tree and specify their internal elements. The color of tree node can be specified using functions and based on type of the room. She uses CONVERt in the same manner as described in the previous section, making use of suggestions, as each of these model structures is large and each example visualized is also large. Carrie can use examples of part of the building model to specify her transformation to a corresponding part of the structure chart. CONVERt then generates a model-to-model translator that can be applied to complete building design models to generate a complete structure chart.

6. Architecture

Our new approach to concrete visualization generation and model mapping generation as presented in this paper has been implemented as a proof of concept in our CONVERt framework [45]. A high-level architecture and key parts of CONVERt are depicted in Figure 13. In the following paragraphs, we briefly describe the implementation of key mechanisms provided by this framework.

The reverse engineering and model abstraction mechanism of CONVERt (Figure 13 (1)) uses a graph lattice as meta-model to be used in transforma-

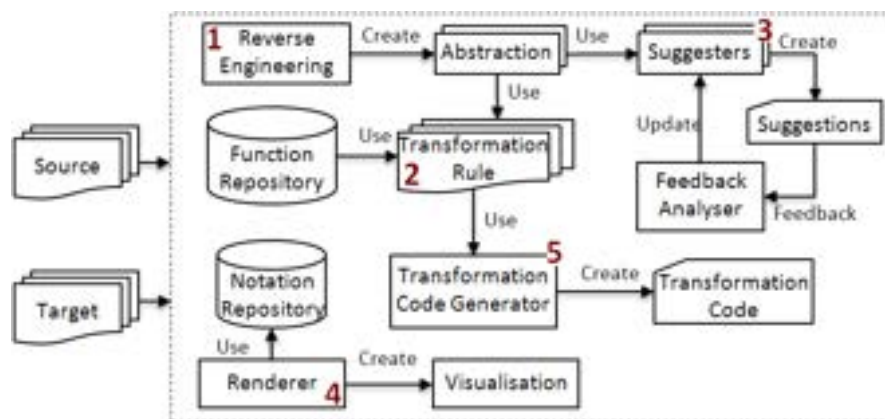


Figure 13: CONVERt's architecture.

tion rules and for scheduling. Once source and target examples are provided, a crawler traverses the examples and fills the lattice with new element structures that it faces. This way the structure of source and target are known to the system. Each transformation rule (Figure 13 (2)) will be initially created by considering a portion of this structure that represents the element being dragged or dropped as a template. As a result, once an element is being dragged, the abstract structure of the input example it represents is also dragged with it. The specified source to target correspondences will be inserted in these templates. Once all correspondences are specified, the transformation code generator (Figure 13 (5)) uses these correspondences and forms XSLT snippets that will be inserted in the template. The altered template will then be used by transformation code generator to generate a transformation rule in XSLT. If an alternative transformation language is needed, translation of these templates to the target languages should be provided to the system.

Each visual notation in CONVERt has a view created and provided by XAML and a Model which is an XML description of the internal elements. A direct mapping is provided to transform notation's model XML to the XAML representation. Since the generated concrete visual notations need to provide interaction (drag and drop) capabilities and host transformation templates, the Renderer mechanism of CONVERt (Figure 13 (4)) wraps each notation in interaction logic provided by an instance of a Visual Element (VE) class. A VE provides a container for the notations and other VEs and is implemented using XAML and C#. This architecture allows our framework to let users interact with composing elements of a model visualization regardless of the embedding hierarchy of the notation.

Our prototype implementation of CONVERt was done using Microsoft Visual Studio and C#. The rendering of XAML based visualizations is native in Windows Presentation Foundation (WPF), available in Visual Studio. Updated versions of CONVERt can be downloaded on-line³.

7. Evaluation

We have used CONVERt to specify a wide range of complex model transformation and information visualization problems. These include several MDE problems using UML source models and code and script targets; CAD tool integration using 2D and tree structure source and target visualizations; and various business analysis problems, including a Minard's Map visualization. Details can be found on CONVERt's website³ including example videos of specifying CONVERt transformations using concrete, by-example visualizations. This section describes our evaluations of CONVERt, which consist of a user study and evaluation of the suggester system.

7.1. User Evaluation

We wanted to get detailed target end user feedback on the CONVERt approach and our prototype concrete, by-example model transformation tool. To do this we designed a user study where users performed a number of model transformation and comprehension tasks with CONVERt.

7.1.1. Participants and Tasks

For our user study, we recruited 19 users (including 4 controls for instrument testing) in two groups from software engineering staff and students at Swinburne University of Technology. Participants were introduced to CONVERt through a 10 minute screen-cast which described the user interface, visualizations and transformation generation procedure. They were then asked to perform a set of given model visualization and mapping tasks and were asked to use a think-aloud approach. The experimental setup comprised a laptop with an attached mouse. Screen captures were taken during the process and a matching questionnaire with 58 questions was handed to each participant at the end of the experiment with 5 point Likert scale (ranging from strongly disagree to strongly agree) and dedicated spaces to leave comments and optional feedback.

³<https://sites.google.com/site/swinmosaic/projects/convert>

The tasks assigned to each group were to create a visualization with CONVERt and then use it as source and generate a transformation from the example visualization to a provided visualization. Both groups had the same settings but used different input examples and target visualization. The first group were given a model representing business sales data and were asked to create a bar chart visualization of their sales data (task 1). They were then asked to transform that bar chart visualization to a pie chart visualization (task 2). The second group were given a class diagram data (XML) and asked to generate a class diagram visualization (task 1). For Task 2 they were asked to transform that class diagram visualization to a provided Java code visualization (similar to usage example of Section 5). Task description hard copies which were handed to the participants did not describe instructional steps. Instead, they included the input file names and their locations, and a snapshot of the desired final visualization and transformation results. Users had to come up with steps required to get similar results. They were allowed to ask questions from the instructor if they had trouble understanding those steps.

Our first group consisted of 10 participants (8 male, 2 female). Second group consisted of 5 participants (3 male, 2 female). In response to demographic questions **D.3**: “How familiar are you with model transformation and modeling in general?” and **D.4**: “How familiar are you with data visualization?”, the participant had following options: **VF**: Very familiar, **SF**: Somewhat familiar, **HH**: Had heard of it, and **NF**: Not familiar. The frequency of responses are provided in Table 1.

Table 1: Partial demographics of participants (%)

Question	NF	HH	SF	VF
D.3	13	33	47	7
D.4	13	20	53	13

7.1.2. Results

Table 2 shows a selection of eight questions from our questionnaire. We have assigned scores of 1 (for perfect negative) to 5 (perfect positive) to each Likert point and calculated the Median, Mode and Frequency of responses. The responses to sample questions based on these arrangements are also summarized in Table 2. Full results can be found on CONVERt’s website³. It took participants on average 29 minutes to accomplish both tasks

successfully. Section 8 provides a discussion of these results. We also asked open ended questions about the use of the tool on these examples, suggestions for improvements, and overall impression of the approach for model transformation problems.

Table 2: Sample questions of the questionnaire.

	Question	Frequency (%)				
		1	2	3	4	5
Q.1	I found it easy to visualize the given data as a bar chart/class diagram.	0	0	20	7	73
Q.2	I learned to use the tool quickly.	0	0	13	27	60
Q.3	Visual diagrams help me better understand the relationships between source and target drawings.	0	0	0	20	80
Q.4	I found it easy to specify the relations between left hand side and right hand side visualizations.	7	0	7	20	67
Q.5	In general I found the tool to be easy for transformation between visualizations.	7	0	0	40	53
Q.6	Recommendations helped me better understand relations between source and target visualizations.	7	0	33	27	33
Q.7	I used recommendations at least once.	7	7	27	20	40
Q.8	I was satisfied with the way recommendations were presented.	7	7	27	13	47

As Table 2 indicates, the majority of participants agree on ease of use of the tool for generating visualizations. This is reflected in their responses to questions **Q.1**. Similarly, the responses indicates that the participants found it easy to learn the tool (see their response to question **Q.2**, 60% strongly agree and 27% agree). As can be seen from results of Table2, users positively responded to having visualizations in better understanding of relationships

between source and target (**Q.3** with 80% strongly agree and 20% agree). These responses demonstrate users' perception of the approach is in accordance to our motivating scenario on usefulness of concrete visual by example approach for generation of mappings. In terms of ease of use, the responses are fairly consistent and indicate general acceptance of the approach and tool-set (see responses to questions **Q.4** and **Q.5**). In response to **Q.4**, total of 87% of the users have agreed that it was easy to specify the relations between left hand side and right hand side visualizations which complements responses to question **Q.3**. Similarly 93% of the users have agreed that the approach provided by the tool for specifying transformation using visualizations was easy (**Q.5**).

The results of Table2 shows potential points of improvement to the approach and specifically to they way recommendations are represented and used. For example, when we asked whether provided recommendations helped users understand relations between source and target visualizations (**Q.6**) only 60% of the users have responded agree and strongly agree. Similarly in question **Q.7**, 60% of participants have agreed that they have used recommendations at least once. In terms of recommendations representation, we have also received 60% satisfaction (in response to **Q.8**) which could be a clue to why users did not use the recommendations and preferred drag and drop to specify correspondences. For example, a participant did not realize that by selecting from suggestions, it is possible to specify correspondences and therefore did not use them at all.

We should also point out that the guide and filter mechanism described in section 5.3 was not implemented in the version of the tool used for evaluation. Users were provided with a list of recommendations instead and to select a recommendation they had to traverse the list to find correspondences. This proved to be problematic and motivated us for implementation of the guide and filter mechanism. We still believe that better representation of recommendations (perhaps by highlighting them in the visualizations) helps improve accessibility and usability of recommendations.

7.2. Recommender evaluation

We wanted to evaluate our CONVERt suggester to see how well it performs in suggesting possible correspondences for large example models. Therefore, we slightly altered its design for this task to be able to evaluate it as a batch model matcher. This modification includes separation of the suggester

Table 3: Categorisation of possible recommendations.

	Recommended	Not Recommended
Relevant	True-Positive (TP)	False-Negative (FN)
Irrelevant	False-Positive (FP)	True-Negative (TN)

system from CONVERt’s GUI and disabling the guide and filter representation mechanism, so that source and target model examples can be processed as a whole. This way, we would be able to evaluate how accurate are the recommended correspondences using a set of available source and target examples and against a benchmark. For this task the suggester was used to check source and a target model examples and provide recommendations for possible correspondences between all elements of the examples. The suggester was used in its initial default setting, i.e. it was configured to give one recommendation per pair and all recommender confidence weights were set to neutral (one in this case) for each set of source/target examples.

We used model and schema matching examples from the Illinois Semantic Integration Archive⁴ to test our CONVERt suggester. We have chosen the house listing information from real estate websites. This selection has been based on availability of examples in the dataset and their intended application, i.e. for testing schema matching techniques. The provided dataset in the archive represent house listings for different websites (e.g. Yahoo and Home Seeker). Each dataset includes set of house listings that are formatted according to the specific website requirements. The datasets do not represent same information, rather each provides a different set of house listings.

To test these examples with CONVERt’s suggester, a correspondence benchmark was developed including all correct correspondences of the examples. Table 3 shows possible categories of recommendations. Using this table, a *Relevant* correspondence is a correspondence available in the benchmark. If this correspondence is recommended, then it is considered as a *True-Positive* correspondence recommendation. If the correspondence is not recommended, it will count as a *False-Negative* correspondence recommendation and so on.

Using the categories of Table 3 and Equations 1 to 3, we calculated Precision (Prec.), Recall (Rec.) and F-Measure (F-M). These metrics were used as they represent most common metrics for evaluating recommender systems

⁴<http://pages.cs.wisc.edu/~anhai/wisc-si-archive/>

[46]. We have applied our suggester system to multiple different combinations of the datasets. For example, suggester system was applied to the problem of matching examples of Yahoo house listings to Home Seeker’s house listings. Table 4 provides results of some of these example evaluations. Other combination of these examples are available on-line³.

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

$$Recall \text{ (True Positive Rate)} = \frac{TP}{TP + FN} \quad (2)$$

$$F\text{-Measure} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (3)$$

Table 4: Evaluation results of the Suggester system.

Example	Default			Optimized		
	Prec.	Rec.	F-M.	Prec.	Rec.	F-M.
NKY - Texas	0.3	0.78	0.44	0.94	0.7	0.8
Yahoo - NKY	0.34	0.73	0.47	0.81	0.6	0.7
Yahoo - Home Seeker	0.44	0.74	0.55	0.92	0.63	0.75
Home Seeker - NKY	0.51	0.86	0.64	0.9	0.81	0.85

Based on Table 4, our Suggester has performed relatively poor on the default setting and for the first calculation of recommendations. For example, the NKY to Texas matching example has achieved precision of 0.3 and recall of 0.78. This indicates that the majority of true recommendations have not been produced. This is due to use of different example-specific naming convention, typing, structure and sizes for the source target example pairs. Also, it was not possible to provide correspondences for certain source and target pairs as in numerous occasions the models did not have corresponding elements. For example, the NKY dataset provides dimensions (given as $X \times Y$) for each listing whereas Texas provides lot size in square meters. Or the NKY includes an element for basement whereas other examples do not provide such elements. Since the suggester gives a correspondence for all pairs, it has had effects on our results.

However, it is possible to configure the suggester system to use certain recommender(s) by using the provided configurations, i.e. users can alter the suggester to use different combination of certain recommenders. Using these configurations and different combinations, it is possible to design an optimization mechanism for the suggester system to optimize its recommendations for better precision and recall. Consecutively possible combinations of recommenders were considered as the search space and the resulted recommendations of different recommender combinations were examined against the benchmarks. The best result was reported as optimized combination configuration. The optimized results are also available in Table 4 and show significant improvements. For example, the precision of NKY to Texas example has improved from 0.3 to 0.94 and its recall has been slightly improved from 0.78 to 0.7. This shows that if previous knowledge or benchmarks of the examples are available, users can customize the suggester system to provide more improved recommendations for certain metrics, e.g. precision.

This optimization is time consuming and may not suit the purpose of our recommender. Also, it may be beneficial to use fuzzy configurations using confidence weights rather than the provided configuration settings, i.e. give a certain recommender less importance (e.g. setting the confidence weight to 0.2) and giving another more credit (e.g. by setting the confidence weight to 0.8). Although this can be achieved by altering the optimization to provide fuzzy confidence values, our experience with CONVERt showed that after continues usage of the recommender (accepting/rejecting) on similar examples, its learning mechanism automatically converges the confidence scores to close to optimized configuration. For example, for the class diagram to Java code example we could achieve Precision of 1.0 after five usage iterations.

8. Discussion

Our user study demonstrated that on average users positively liked the idea of concrete visual transformation. See for example their answers to question **Q.3** in Table 2 where all users highly rated (4 or 5) the use of visual diagrams in understanding the source-to-target relationships and question **Q.4** where 87% of the users rated the ease of use of the approach high or very high (67% very high and 20% high). However, certain drawbacks of the used version of prototype tool affected user experience. Specifically, some users could not differentiate model elements and placeholders as they were represented similarly for each visual notation by earlier versions of the

framework. This resulted in confusion, and a user had to ask the instructor after a mistake was made in notation composition. Our tool support for CONVERt was updated to reflect this and show place holders and notation elements separately (see for example compositions of Figure 6).

In comparison, our suggester mechanism achieved a lower user acceptance than the visualization and transformation specification. This has been reflected in users responses to questions **Q.7** and **Q.8** in Table 2. Hence, our third research question - "Can we provide guidance to users with visual representations and recommendations for large model mapping problems?" - is not completely addressed. We believe this is due to the above mentioned representation of recommendations and the fact that the given examples were sufficiently simple; therefore users already knew most of the correspondences, and thus did not need to utilize its potential most of the time. For example a participant stated that the mapping correspondences were "easy to find and specify" and therefore felt no need to use them. Provided that the visualizations were more complex, it would have evaluated effects of the suggestion much better. One participant did not realize that by selecting from suggestions, it is possible to specify correspondences and therefore did not use the recommendations. Also, the way recommendations are currently presented was not well received by users and some users found it hard to find the presented recommendations in visualizations and accordingly accept or reject them. This is potentially due to current use of element hierarchy in representation of left hand side and right hand side elements. For example a UML class would be represented as *ClassDiagram/Class/UMLClass*. As a result better representation of recommendations is being considered for future CONVERt versions using interactive highlighting in visualizations [14].

While writing transformation code, a transformation designer might partition the code into several modules (e.g. transformation rules) or might write the transformation as one module. It is a common software engineering practice to modularize the code to help better readability and easier maintenance. Although readability of the automatically generated transformation code in our approach was not considered in the design, it demonstrates an acceptable modular structure, i.e. the code is divided into separate transformation rules (in this case XSLT templates). This is due to the fact that each notation-to-notation mapping is considered as one transformation rule template. As a result, if need be to reuse the generated code outside framework, it exhibits acceptable readability specially for large transformations.

8.1. Threats to validity

Internal: Four of our participants mentioned the effect of learning during the experiments. They admitted that since the drag and drop tasks were being repeated for tasks one and two, they could perform the second task easier. This might have had effects on better acceptance of the approach for task two. Some participants may have been reluctant to ask questions regarding the items being asked in the questionnaire and therefore responded based on their understanding of the questions.

Our suggester system uses the results returned by name similarity recommender when deciding which neighbors are similar. As a result accuracy of structural and propagated similarity recommenders are dependent on name similarity. This can have effects on their accuracy in situation where names of source and target elements are not similar.

External: The users whom participated in the evaluation were mostly chosen among staff and students of Swinburne University of Technology (18 out of 19). This represents a bias and will affect generalization of our claims. Also 47 percent of the participants shared a common background in Software Engineering and 40 percent shared a background in computer science. As a result, their background could have introduced bias in terms of their familiarity with software tools. However, given our target end user community is predominantly such engineers and technical experts, some generalization is reasonable.

The examples used for evaluating our suggester system were from schema matching test cases and therefore were not completely serving the purpose of evaluating a recommender system for model transformation domain. This could have affected our suggester system evaluation.

Construct: Due to simplicity of the experiment for one group, performing bar chart to pie chart transformation, five participants did not use the recommendations. These have had effects on evaluation of the recommendation system. Also, some instructions made to the participants by the instructor during experiment may have affected the participants' experience. The instructor was asked not to give any instructions unless asked by the participants. Our observation of the responses and the recordings, demonstrated that the participants who requested more instructions had accordingly mentioned this need in their responses.

With regards to our suggester evaluation and as stated with external validity, the nature of the tested examples might have affected the construct

validity of our suggester evaluation. We will keep looking and designing experiments and examples to be able to better evaluate the use of recommendations in model and visualization transformation domain.

Statistical: It is possible that the inferences we have made from our results are due to limited number of participants. The statistics we have used are calculated having non-parametric characteristics of the responses in mind. We do not reject the possibility of changes in the inferences when the number of users increases.

The precision, recall and F-Measure metrics used in evaluating the suggester system are very much dependent on the benchmark they are being tested against. This benchmark was generated based on all available correspondences. In some cases, there are multiple correct suggestions for example when referring to size of a room as height and width vs. Square meters, both height and width can be considered as correct correspondences for size. Given that the Suggester system provides only one suggestion per pair by default, lots of such multi possibilities are not considered. This has resulted in lower number of true-positive choices and higher number of false-negatives and consecutively lower precision and recall.

8.2. Future work

Although model mappings indicated by correspondences are often bidirectional i.e. mapping information from the target back to the source, it is not always possible to achieve bi-directionality. For example, when using a function to add two values and dragging the output onto an element, it is not possible to directly generate the values in reverse unless at least one of the original values are saved. We call these “Lossy” transformations as the information for creating the forward transformation is lost during the process. Addressing such transformations defines part of our future work.

Our main goal in designing the CONVERt approach was to provide a concrete visual approach to specify model transformations, rather than ensuring the completeness and correctness of transformations (e.g. see [47]). It is possible to check the correctness of the resulted target visually (by checking how the target is rendered), adding constraint checks to notations, or by using meta-models to check conformance of the generated targets. The correctness of the generated transformation still remains an open future avenue in this visual by example approach. Other areas of key future work include applying the CONVERt approach to other model-based domains including information visualization, tool integration, and EDI and XML message translation,

further user studies of the approach perhaps by practitioners, different visualization implementation (e.g. SVG) and different target transformation language generation (e.g. ALT or QVT vs. XSLT).

9. Summary

We have presented a new approach for transformation generation using familiar concrete visualizations of source and target model examples. Through use of these visual notations, the required knowledge and skill for performing model transformation specification is reduced. The system presented in this paper provides abstractions by reverse engineering model examples and gives users the capability to specify correspondences on familiar notations or use correspondences suggested by the system. This approach is capable of generating comprehensive model transformers for a wide variety of applications. We have evaluated our approach and its tool support in a user study and the results provide general acceptance of the approach in using drag and drop approach for visualization and use of concrete visualizations for transformation between two visualizations.

Acknowledgments

We thank anonymous reviewers for their comments and feedback. This work was supported in part by Australian Research Council Discovery Project (DP140102185), Swinburne University of Technology (Australia) Early Research Career grant, and German Research Foundation (DFG) under the Priority Program SPP1593: Design For Future - Managed Software Evolution.

References

- [1] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Conceptual modelling and its theoretical foundations, Springer-Verlag, 2012, Ch. Model transformation by-example: a survey of the first wave, pp. 197–215.
- [2] Y. Sun, J. White, J. Gray, Model transformation by demonstration, in: A. Schürr, B. Selic (Eds.), Model Driven Engineering Languages and Systems, Vol. 5795 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 712–726.

- [3] M. Faunes, H. Sahraoui, M. Boukadoum, Generating model transformation rules from examples using an evolutionary algorithm, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, ACM, New York, NY, USA, 2012, pp. 250–253.
- [4] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, M. Roth, Clio grows up: from research prototype to industrial tool, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 805–810.
- [5] A. Raffio, D. Braga, S. Ceri, P. Papotti, M. Hernandez, Clip: a visual language for explicit schema mappings, in: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, 2008, pp. 30–39.
- [6] D. Varró, Model transformation by example, in: Model Driven Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 410–424.
- [7] M. Kessentini, H. Sahraoui, M. Boukadoum, O. B. Omar, Search-based model transformation by example, *Software & Systems Modeling* 11 (2012) 209–226.
- [8] H. Ehrig, U. Prange, G. Taentzer, Fundamental theory for typed attributed graph transformation, in: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), *Graph Transformations*, Vol. 3256 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 161–177.
- [9] G. Rozenberg, H. Ehrig, *Handbook of graph grammars and computing by graph transformation*, Vol. 1, World Scientific London, 1999.
- [10] A. Schürr, Specification of graph translators with triple graph grammars, in: E. Mayr, G. Schmidt, G. Tinhofer (Eds.), *Graph-Theoretic Concepts in Computer Science*, Vol. 903 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1995, pp. 151–163.
- [11] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

- [12] S. Hidaka, Z. Hu, H. Kato, K. Nakano, A compositional approach to bidirectional model transformation, in: *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, 2009, pp. 235–238.
- [13] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer, Information preserving bidirectional model transformations, in: *Proceedings of the 10th international conference on Fundamental approaches to software engineering, FASE'07*, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 72–86.
- [14] S. Bossung, H. Stoeckle, J. Grundy, R. Amor, J. Hosking, Automated data mapping specification via schema heuristics and user interaction, in: *Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering.*, 2004, pp. 208–217.
- [15] B. Shneiderman, Direct manipulation: A step beyond programming languages (abstract only), in: *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems. (Part - II): Human Interface and the User Interface - Volume 1981, CHI '81*, ACM, New York, NY, USA, 1981, pp. 143–.
- [16] J. Grundy, R. Mugridge, J. Hosking, P. Kendall, Generating edi message translations from visual specifications, in: *Proceedings of the 16th IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 2001, pp. 35–42.
- [17] Y. Li, J. Grundy, R. Amor, J. Hosking, A data mapping specification environment using a concrete business form-based metaphor, in: *IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, IEEE, 2002, pp. 158–166.
- [18] I. Avazpour, Towards user-centric concrete model transformation, Ph.D. thesis, Swinburne University of Technology (2014).
- [19] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, M. Wimmer, Lifting metamodels to ontologies: A step to the semantic integration of modeling languages, in: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (Eds.), *Model Driven*

Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 528–542.

- [20] F. Budinsky, Eclipse modeling framework: a developer's guide, Addison-Wesley Professional, 2004.
- [21] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, Metamodel matching for automatic model transformation generation, in: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, MoDELS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 326–340.
- [22] K. Voigt, T. Heinze, Metamodel matching based on planar graph edit distance, in: Proceedings of the Third international conference on Theory and practice of model transformations, ICMT'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 245–259.
- [23] L. Lafi, S. Hammoudi, J. Feki, Metamodel matching techniques in mda: challenge, issues and comparison, in: Proceedings of the First international conference on Model and data engineering, MEDI'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 278–286.
- [24] M. Siikarla, A Light-weight Approach to Developing Interactive Model Transformations, Phd thesis, Tempere University of Technology (2011).
- [25] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, P. Buono, Research directions in data wrangling: Visuatizations and transformations for usable and credible data, *Information Visualization* 10 (4) (2011) 271–288.
- [26] R. Lämmel, E. Meijer, Mappings make data processing go 'round, in: Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering, GTTSE'05, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 169–218.
- [27] R. Grønmo, Using concrete syntax in graph-based model transformations, Ph.D. thesis, University of Oslo (2009).
- [28] H. Stoeckle, J. Grundy, J. Hosking, Approaches to supporting software visual notation exchange, in: *Human Centric Computing Languages and*

- Environments, 2003. Proceedings. 2003 IEEE Symposium on, 2003, pp. 59–66.
- [29] H. Stoeckle, J. Grundy, J. Hosking, A framework for visual notation exchange, *J. Vis. Lang. Comput.* 16 (3) (2005) 187–212.
- [30] L. Grunske, L. Geiger, M. Lawley, A graphical specification of model transformations with triple graph grammars, in: A. Hartman, D. Kriesche (Eds.), *Model Driven Architecture Foundations and Applications*, Vol. 3748 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 284–298.
- [31] R. Grnmo, B. Mller-Pedersen, G. Olsen, Comparison of three model transformation languages, in: R. Paige, A. Hartman, A. Rensink (Eds.), *Model Driven Architecture - Foundations and Applications*, Vol. 5562 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2009, pp. 2–17.
- [32] Z. Hemel, L. Kats, E. Visser, Code generation by model transformation, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), *Theory and Practice of Model Transformations*, Vol. 5063 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 183–198.
- [33] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced concepts and tools for in-place emf model transformations, in: *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010*, Vol. 6394 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 121–135.
- [34] F. Chauvel, J.-M. Jézéquel, Code generation from uml models with semantic variation points, in: L. Briand, C. Williams (Eds.), *Model Driven Engineering Languages and Systems*, Vol. 3713 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 54–68.
- [35] I. Stürmer, M. Conrad, Test suite design for code generation tools, in: *Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering*, 2003, pp. 286–290.
- [36] U. Nickel, J. Niere, A. Zündorf, The fujaba environment, in: *Proceedings of the 22nd international conference on Software engineering, ICSE '00*, ACM, New York, NY, USA, 2000, pp. 742–745.

- [37] M. Ganapathi, C. N. Fischer, J. L. Hennessy, Retargetable compiler code generation, *ACM Comput. Surv.* 14 (4) (1982) 573–592.
- [38] G. S. Swint, C. Pu, G. Jung, W. Yan, Y. Koh, Q. Wu, C. Consel, A. Sahai, K. Moriyama, Clearwater: extensible, flexible, modular code generation, in: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05*, ACM, New York, NY, USA, 2005, pp. 144–153.
- [39] J. Huh, J. Grundy, J. Hosking, K. Liu, R. Amor, Integrated data mapping for a software meta-tool, in: *Proceedings of the 2009 Australian Software Engineering Conference, IEEE*, 2009, pp. 111–120.
- [40] I. Avazpour, J. Grundy, H. Vu, Generating reusable visual notations using model transformation, in: *7th International Symposium on Visual Information Communication and Interaction (VINCI)*, 2014, pp. 58–67.
- [41] R. Singh, J. Xu, B. Berger, Global alignment of multiple protein interaction networks with application to functional orthology detection, *Proceedings of the National Academy of Sciences* 105 (35) (2008) 12763–12768.
- [42] D. Gusfield, R. W. Irving, *The stable marriage problem: structure and algorithms*, Vol. 54, MIT press Cambridge, 1989.
- [43] F. Hernández del Olmo, E. Gaudioso, Evaluation of recommender systems: A new approach, *Expert Syst. Appl.* 35 (3) (2008) 790–804.
- [44] R. Amor, G. Augenbroe, J. Hosking, W. Rombouts, J. Grundy, Directions in modelling environments, *Automation in Construction* 4 (3) (1995) 173–187.
- [45] I. Avazpour, J. Grundy, CONVERt: A framework for complex model visualisation and transformation, in: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 237–238.
- [46] I. Avazpour, T. Pitakrat, L. Grunske, J. Grundy, *Recommendation systems in software engineering*, Springer, 2014, Ch. Dimensions and Metrics for Evaluating Recommendation Systems, pp. 245–273.

- [47] H. Kargl, M. Wimmer, Smartmatcher – how examples and a dedicated mapping language can improve the quality of automatic matching approaches, in: International Conference on Complex, Intelligent and Software Intensive Systems., 2008, pp. 879–885.

4.3 SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions

Grundy, J.C., Cai, Y. and Liu, A. SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions, *Automated Software Engineering*, Kluwer Academic Publishers, vol. 12, no. 1, January 2005, pp. 5-39

DOI: [10.1023/B:AUSE.0000049207.62380.74](https://doi.org/10.1023/B:AUSE.0000049207.62380.74)

Abstract: Most distributed system specifications have performance benchmark requirements, for example the number of particular kinds of transactions per second required to be supported by the system. However, determining the likely eventual performance of complex distributed system architectures during their development is very challenging. We describe SoftArch/MTE, a software tool that allows software architects to sketch an outline of their proposed system architecture at a high level of abstraction. These descriptions include client requests, servers, server objects and object services, database servers and tables, and particular choices of middleware and database technologies. A fully-working implementation of this system is then automatically generated from this high-level architectural description. This implementation is deployed on multiple client and server machines and performance tests are then automatically run for this generated code. Performance test results are recorded, sent back to the SoftArch/MTE environment and are then displayed to the architect using graphs or by annotating the original high-level architectural diagrams. Architects may change performance parameters and architecture characteristics, comparing multiple test run results to determine the most suitable abstractions to refine to detailed designs for actual system implementation. Further tests may be run on refined architecture descriptions at any stage during system development. We demonstrate the utility of our approach and prototype tool, and the accuracy of our generated performance test-beds, for validating architectural choices during early system development.

My contribution: Developed initial ideas for this research, co-designed approach, wrote some of the software the approach based on, co-supervised Masters student, wrote majority of the paper, co-lead investigator for funding for this project from FRST

SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions

John Grundy^{1,2}, Yuhong Cai² and Anna Liu³

¹Department of Electrical and
Computer Engineering and
²Department of Computer Science,
University of Auckland
Private Bag 92019, Auckland
New Zealand
john-g@cs.auckland.ac.nz

³Software Architectures and Component Technologies
CSIRO Mathematical and Information Sciences,
Locked Bag 17, North Ryde, NSW 1670, Sydney
Australia
Anna.Liu@cmis.csiro.au

Abstract

Most distributed system specifications have performance benchmark requirements, for example the number of particular kinds of transactions per second required to be supported by the system. However, determining the likely eventual performance of complex distributed system architectures during their development is very challenging. We describe SoftArch/MTE, a software tool that allows software architects to sketch an outline of their proposed system architecture at a high level of abstraction. These descriptions include client requests, servers, server objects and object services, database servers and tables, and particular choices of middleware and database technologies. A fully-working implementation of this system is then automatically generated from this high-level architectural description. This implementation is deployed on multiple client and server machines and performance tests are then automatically run for this generated code. Performance test results are recorded, sent back to the SoftArch/MTE environment and are then displayed to the architect using graphs or by annotating the original high-level architectural diagrams. Architects may change performance parameters and architecture characteristics, comparing multiple test run results to determine the most suitable abstractions to refine to detailed designs for actual system implementation. Further tests may be run on refined architecture descriptions at any stage during system development. We demonstrate the utility of our approach and prototype tool, and the accuracy of our generated performance test-beds, for validating architectural choices during early system development.

1. Introduction

Most system development now requires the use of complex distributed system architectures and middleware [1, 35]. Architectures are quite varied and systems may use simple 2-tier clients with a centralised database; 3-tier client, application server and database divisions of responsibility; multi-tier, decentralised web, application and database server layers; and peer-to-peer communications [1, 33, 35]. Using each of these approaches an architect has a wide range of choice as to where to locate data processing (e.g. client-side or server-side), whether to cache data (e.g. in an application sever to reduce database overheads), and whether to use combinations of architectural styles for different parts of the same overall system. Middleware used to connect distributed components of a system may include socket (text and binary protocols), Remote Procedure (RPC) and Remote Method Invocation (RMI), DCOM and CORBA, HTTP and WAP, and XML-encoded data [10, 27]. Data management may include relational or OO databases, persistent objects, XML storage, files. Integrated solutions combining several of these approaches, such as J2EE and .NET, are becoming increasingly common [32].

Typically system architects need to work within quite stringent performance (and other) quality requirements that systems implemented using their designs must eventually meet. For example, a system might need to be able to handle a specified maximum number of concurrent users with a particular user request satisfied in a maximum specified time. However, it is very difficult for system architects to determine appropriate architecture organisation, middleware and data

management choices that will meet such requirements during architecture design [27, 13]. Very often architects make such decisions based on their prior knowledge and experience. In order to enable developers to get a more accurate assessment of architectural design decisions various approaches have been developed to validate architectural designs. These include architecture-based simulation and modelling [4, 24, 36], performance prototypes [19, 10, 17], and performance monitoring and visualisation of similar, existing systems [4, 34]. Simulation of architecture and middleware performance tends by its very nature to be more or less inaccurate, giving results of limited usefulness. Performance prototypes may give much more accurate estimates of implemented system performance but require considerable development effort to build and any evolution of architecture design means new or substantially modified prototypes need to be developed. Analysing existing system performance through monitoring requires close similarity between the existing system and proposed system and often considerable modification may be required to gain any useful performance results.

We have developed SoftArch/MTE, an integrated tool allowing software architects to accurately test the performance of their architecture designs by generating performance test-beds from high-level architecture design diagrams. Architects sketch high-level system descriptions, including client, server, database and host elements, and expected client requests and server and database services using a visual Architecture Description Language. From these descriptions SoftArch/MTE automatically generates a runnable, multiple client and server deployable performance test-bed. This test bed code incorporates dummy code to generate client, server and database requests using the specified middleware and data management approaches and adheres to the specified architectural organisation. Our generated performance test beds are deployed and run on multiple client and server hosts, providing a realistic deployment scenario for the code. Generated code and deployed application annotations automatically capture performance measurements as the test bed code runs. Recorded performance data is then relayed to SoftArch/MTE where the data is displayed in the high-level architecture diagrams and by using external data visualisation tools such as MS Excel™. Results from different test runs, architecture organisations and middleware and data management choices can be compared.

We first motivate our research with a simple distributed system development example, an on-line video system, back grounding the two previous research projects we have combined to produce the SoftArch/MTE tool. We critique related work into architecture and middleware performance evaluation techniques and then give an overview of the key elements of our SoftArch/MTE approach. We illustrate how an architect constructs a high-level architecture description using our currently-supported architecture meta-model elements. We describe and illustrate our extensible code generation approach that use XML and XSLT transformations to generate complex performance test bed code. We describe and illustrate the test bed deployment, performance testing and result visualisation process we have developed. We conclude with an analysis of the utility of SoftArch/MTE for performance analysis of software architecture designs and with a summary of our main research contributions and directions for future work.

2. Motivation

All distributed systems have a variety of non-functional constraints that an implementation of the system should adhere to, for example security, data integrity, reliability, performance, exception handling, user interface approach and so on [2, 5]. One of the driving factors in choosing a particular architectural organisation and communications middleware for a system is to ensure these constraints can be met by a design and implementation derived from this architecture [10, 2, 33]. One of the most difficult constraints to ensure is met during architecture (and system) design is performance. This is due to a number of factors: insufficient knowledge of detailed design decisions; unknown performance characteristics of many possible middleware and database choices; unknown performance characteristics of new and often existing software on particular hardware and network platforms; and the complexity of the design meaning combinations of architecture, design and implementation choices may never have been tried by the development team before [2, 10]. This makes validating architectural design decisions in terms of them meeting required system performance requirements virtually impossible during early design.

Consider the development of a system to support an on-line video store library [12], supporting customer on-line video search/reservation and staff in-store video rental management tasks. Some example interfaces for such a system are illustrated in Figure 1 (a). Just one of the many possible architectures that could be used to implement this system is shown in Figure 1 (b). In this example, video store staff use desktop applications connecting to the database(s). Customers interact with user interfaces that connect to application servers, in turn connecting to possibly other servers and one or more databases e.g. holding staff, customer, video, video rental etc details. Data processing may be centralised or spread across clients or servers. Middleware may be HTML, Java RMI, DCOM, CORBA, or XML. Server objects may be COM, CORBA or Enterprise Java Beans. Data management may use relational, object or XML databases, or files.

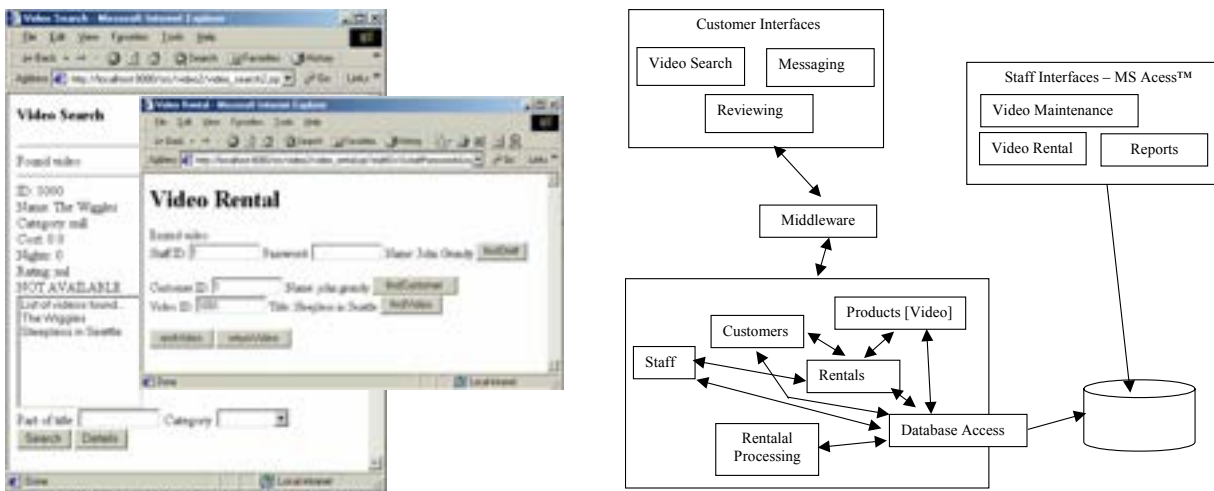


Figure 1. Parts of a simple on-line video system.

The software architect typically has some performance constraints from a non-functional analysis for the system that any chosen architecture design must meet. For example, such constraints might include maximum number of users, response time for different user requests and data processing services, and so on. There may possibly be some further constraints that the architect needs to work within, for example some hardware and software constraints e.g. must run on Windows/LINUX machines; must run on low-end desktop machine; must run over 56kbps modem connection; must use either CORBA or DCOM middleware protocol, must use SQL Server™ 7 database, and so on. In addition, if a system under design must interact with existing systems as part of their operation then the performance and other constraints of the existing applications and their deployment will impact the performance of parts of the new system.

To determine a suitable architecture to use for a system, and appropriate middleware and database choices, an architect typically relies on past experience. They often use rapid prototyping or simulation tools to verify that design choices will likely result in architectural non-functional constraints being met (e.g. performance, number of concurrent users, reliability and so on). Previously developed benchmarks for particular architectures and middleware choices can also assist in guiding design. In this paper we describe an approach to generating performance test-beds (essentially rapid prototypes) directly from a software architecture description. The aim is to enable architects to more quickly and iteratively understand the performance impacts of their architecture-level design decisions. This paper provides a more detailed description of the work previously reported in [15], and includes a number of extensions to the architecture modeling, code generation and evaluation results.

3. Related Work

The need to evaluate software architecture and distributed systems middleware performance has been long recognised by researchers and practitioners [17, 27, 13, 7]. The main approaches to assessing architecture and middleware performance have been the development of test beds (prototypes), the use of simulation and other modelling techniques to estimate likely performance of architectures, and the use of performance monitoring technologies to assess performance of developed systems. The first two approaches are usually used to try and understand likely performance of an architecture before constructing a full system or substantial part of a system based on it. The last approach may be used either before designing a new system similar to an existing one being monitored or may be used after having developed a system and then needing to gain better understanding of aspects of its architecture in order to improve its performance.

Developers typically build prototype performance test beds by hand [10, 17]. This approach is usually very time-consuming for all but the most simple of architectures [10]. If middleware selection is changed or if even simple re-organisation of the architecture design is made very often a whole new test bed prototype needs to be built to assess the likely performance of the new architecture. If when a prototype was first developed it failed to capture adequately the kinds of client/server operations or division of responsibilities that end up being embodied in an evolving architecture design, repeated modifications or replacement of the prototype are necessary. Given that most performance prototypes are in many ways fairly mechanical to produce [17] it would be desirable to generate such code from higher level models of software architectures with the generated test bed code able to be easily regenerated when these models are changed.

Benchmarks published for various middleware and database systems can be useful for architects to gauge possible relative performance for different design choices [7, 10]. These are usually obtained from common benchmarking programs – basically performance test-bed prototypes developed for general use so results of technologies and vendor solutions can be directly compared. However most real system architectures are a complex mix of design choices (architecture layout and divisions of responsibility; middleware and database choices; and host machine and network characteristics). Accurate performance measures for a particular architecture design thus can only be gained from a prototype sufficiently close to the intended eventual system.

In order to avoid having to build continuously evolving performance test bed prototypes a number of architecture and middleware performance simulation and modelling methods and tools have been developed [1, 9, 7, 18, 27, 24, 29, 36]. These make it easier for architects to express and explore likely architecture performance by providing higher-level models of architectures and using these to produce simulated or computed performance estimates for the models. All of these approaches can only provide estimates of possible architecture model performance, however, as they factor out a variety of performance-impacting attributes e.g. some host computer and network characteristics, performance of particular versions of application software and so on [26]. In addition, the accuracy of the performance estimates are highly dependent on the details of the model used and the ability of the simulation and modelling techniques used [18, 27, 22, 36]. This leads to performance results derived from abstract architecture model analysis possibly being quite inaccurate. In addition, the specification of architecture characteristics and the visualisation of simulation-derived performance measures are often predominantly text-based. These approaches provide software architects with low-level detail about architecture performance estimates requiring further analysis and visualisation.

The performance of deployed distributed systems has been of interest to developers for many years and has resulted in the development of a wide variety of middleware, network, database and software performance monitoring and architecture visualisation tools [27, 3, 8, 16, 34]. Many of these approaches aim to provide high-level abstractions for viewing architecture structure and/or performance results [8, 16]. While Visualisations in many of these tools are often quite low-level i.e. tend to focus on low-level programmatic features rather than high-level architectural abstractions, some monitoring tools capture low-level performance information and aggregate this and present it to the user in a higher-level abstraction [34, 27]. Most however do not present performance results in a form compatible with the same visual abstractions used when modelling architecture designs making interpretation of results more difficult. Most performance monitoring techniques require a fully developed system in order to be used. During architecture design the only way to get such systems to evaluate are to build prototypes or try and evaluate similar systems to the one under development. The former approach means much effort particularly if the design continues to evolve and the later is only useful if the two systems are very similar in likely transaction mix and structural organisation.

Many researchers have investigated ways of visualising software architecture structure and behaviour. These range from static architecture modelling approaches usually focusing on structure and to a lesser degree behaviour [31, 20, 23, 25] to dynamic architecture visualisation and metrics analysis [8, 36, 24, 11]. We chose to use our SoftArch architecture modelling and analysis tool in this research as it provides a mix of both static (structure and behaviour modelling) and dynamic (performance and other architecture metric visualisation) support facilities within one integrated environment [12].

4. Background: MTE and SoftArch

The Middleware Technology Evaluation (MTE) project at the CSIRO [5] has been working for several years on developing benchmarks for a wide variety of software architectures and associated middleware and database technologies. An additional aim of this work has been to better understand the impact of various middleware approaches and implementation technologies on system performance [10]. This work has included assessing the performance of systems built using Enterprise Java Beans, DCOM, MQ Series, and Java Messaging Service, and assessing the relative performance of different vendor solutions using each of these technologies. The project team has built many performance “test beds” that have been used to gather performance metrics for a wide variety of middleware technologies and vendor solutions. These test beds are simple distributed system implementations designed to extensively benchmark performance of using basic distributed system architectural approaches implemented with particular middleware technologies.

Figure 2 (a) shows an example MTE test-bed architecture used to evaluate the performance of basic J2EE-based enterprise applications. A simple order processing application was built using J2EE technology and some performance testing clients and server instrumentation code used to gather performance results [5, 10]. Different J2EE server and database implementations can be tried out with the test-bed to measure their relative performance. Multiple client

instances on different host machines can be used or a single client with multiple threads. Other MTE test-beds use different architecture choices e.g. J2EE Bean-managed vs. Container-managed persistence, and make use of different middleware technologies e.g. CORBA, MQ Series and .NET.

Figure 2 (b) shows some simplified pseudo-code examples for example clients and servers used in the J2EE order processing test-bed. The client(s) runs one or more requests on the server EJBs and measure their performance. The clients can be quite complex, mixing a range of different server requests and reporting their performance. The MTE clients can also be configured to run requests on the server with no pause, to measure total raw transaction throughput obtainable, or with pausing between server requests to simulate user interactions with clients.

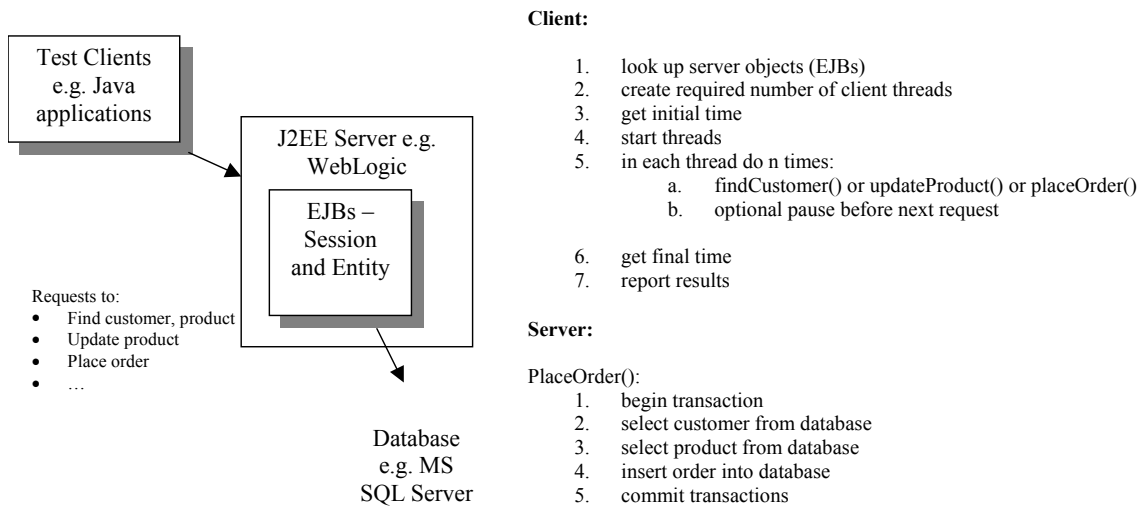


Figure 2. (a) Example of test-bed architecture; and (b) some partial test-bed code examples.

The results of these test-bed performance evaluations provide developers assistance in identifying the general performance characteristics of a range of basic software architectures using different middleware and database technology choices. However, a major problem of this approach is that in order to assess each middleware approach/vendor solution for different architecture designs, even quite simple ones, new prototype test beds must be developed. This is very time-consuming. In addition, these test beds only assess the deployment of the middleware solutions in simple, exemplar architectures, such as the simple on-line order processing example above. Developers must still extensively develop prototypes with additional details about their planned system architecture in order to get an accurate understanding of this architecture's likely performance.

In a different project we developed a tool, SoftArch, for designing complex software architectures, generating partial object-oriented designs and visualising fully-implemented system architecture performance [12]. This tool allows software architects to model high-level architecture designs using a visual language and refine their designs using successive levels of abstraction into design-level classes. Figure 3 shows SoftArch being used to model a candidate architecture design for the video system in Figure 1 [12]. SoftArch provides a variety of predominantly graphical architecture modelling tools (1) and an extensible meta-model of available architecture elements, connectors and properties. It also provides a set of "design critics" (2) that monitor software architecture model changes and give unobtrusive user feedback. Data collected by performance monitoring annotations in code developed from SoftArch models is used to visualize running system performance at high-levels of abstraction using SoftArch diagrams (3), showing the user relative time consumption by architecture components as well as loading on network connections, object creation metrics and so on.

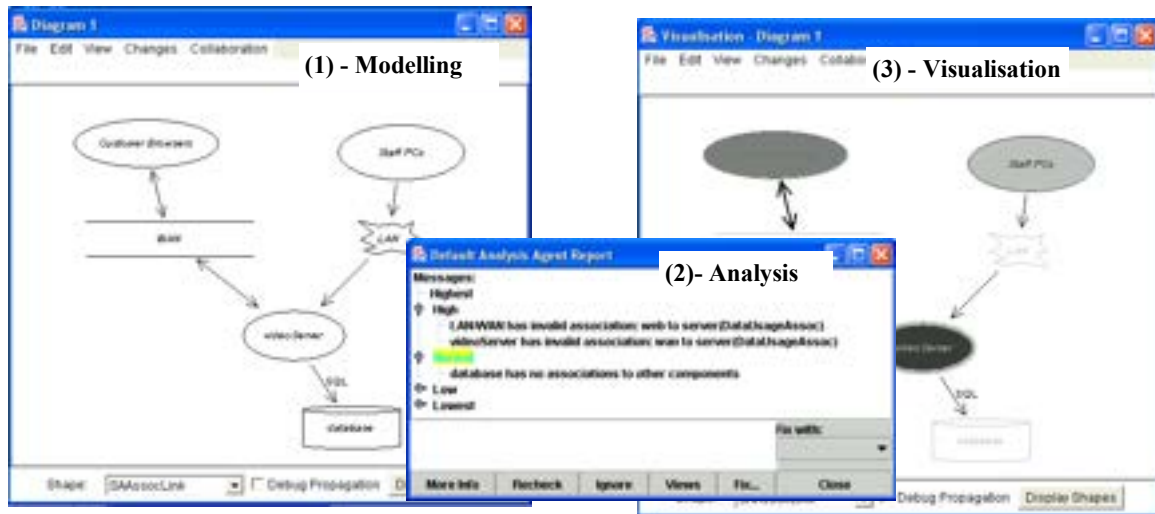


Figure 3. Example of SoftArch in use.

We wanted to try and unify our MTE work and our SoftArch work to see if we could take SoftArch architecture specifications, which include key architectural abstractions, middleware choices, client and server deployment information and so on, and generate MTE-style performance test-beds directly from these high-level architectural models. From the various MTE test-beds that had been developed [5], we recognized that almost all of these test beds could indeed be automatically generated and run from SoftArch models. To achieve this we took SoftArch's saved architecture models and used them as the input for a code generator producing MTE-style performance test bed code and batch control scripts. Our new SoftArch/MTE environment generates MTE-style performance test beds from SoftArch architecture design diagrams, automatically runs multiple tests and captures performance measures, and visualises these results back in SoftArch diagrams. The aim of this work was to fully-automate MTE-style performance test bed generation, deployment of generated code and scripts, and capture and presentation of performance results for software architects. This environment supports a rapid, exploratory architecture design process where architects can easily make changes to architectural design and middleware choices and from these high level system descriptions receive accurate estimates of the eventual, fully-developed system performance. This approach aims to much reduce architecture validation time and improve eventual developed system architecture quality from a performance perspective.

5. The SoftArch/MTE Performance Evaluation Process

Performance results obtained from standard benchmarking suites and MTE-style prototypes provide accurate estimates of architecture performance as they run using real code on real machines. Similarly analysis of the performance of real systems via monitoring obtains accurate results as to architectural component behaviour when a system embodying an architecture design is deployed. However both of these approaches suffer from the effort to build performance prototypes or are only useful after a system has been implemented. In contrast the use of performance simulation of high-level models of architectures allows architects to easily specify and evolve the architecture specification but at the cost of having to accept often very imprecise simulation results due the huge complexity in modelling all performance-impacting factors on a real system's performance. SoftArch/MTE adopts the high-level specification and results-presentation approach of simulation-based approaches. However it automatically generates real executable performance test bed code that is deployed on real client and server machines and is monitored to obtain accurate real-system performance results.

The SoftArch/MTE prototype environment supports integrated, evolutionary architecture modelling, test bed generation, performance analysis and evolution. Figure 4 outlines the process whereby software architects obtain performance results from real code generated by SoftArch/MTE. Steps 2-6 are fully automated. The architect first constructs a high-level architecture design, specifying clients, servers, remote server objects and database tables, client-server, server-server, client/server-database requests and server services, and various kinds of connectors between these architectural abstractions (belongs-to, runs-on, network connection, etc) (1). They also specify various properties: client, server and database host machine; number and frequency of requests (e.g. 1000 times; continuous; every 0.25 seconds; etc); database table and request complexity (e.g. one row select; 100 row select/update; one row insert/delete etc);

middleware protocol (e.g. CORBA using Visibroker 4.0; TCP/IP socket using textual XML document; etc); and so on. Available modelling elements and their properties are specified in an extensible SoftArch/MTE meta-model.

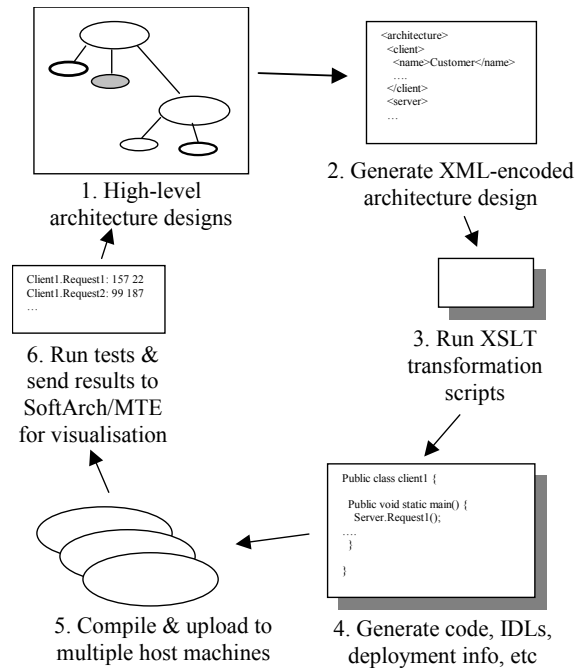


Figure 4. SoftArch/MTE architecture analysis process.

The architect instructs SoftArch/MTE to generate an XML encoding of the architecture model (2). This is then passed through a number of XSLT (XML style sheet transformations) scripts (3), which generate Java, C++, Delphi etc code, along with CORBA and COM IDL files, EJB deployment descriptors, database table creation and population scripts, compilation and start-up scripts, and so on (4). This generated code is a fully working performance test-bed, compiled without architects viewing or editing it (5). Compiled (deployable) client and server program code is then uploaded to the specified client, server and database host machines by sending them to a deployment agent running on those machines via Java RMI. The generated client and server programs and appropriate database servers are started on all hosts and clients wait for a SoftArch/MTE signal (via their deployment agent), or a scheduled start time, to begin execution i.e. sending requests to servers. SoftArch/MTE client and server code annotations are usually used to capture performance measures, though we can also generate scripts to configure performance monitoring programs to collect performance measures. Once tests complete, deployment agents collect results (usually from client and server program output files) and send these to SoftArch via RMI (6). SoftArch annotates architecture diagrams in various ways to highlight the performance measures captured from running the generated test beds. SoftArch/MTE can also generate performance summary analysis and invoke a 3rd party data visualisation tool to show performance details and summary charts (we use MS Excel™ to do this). Multiple test run results using different middleware, databases and client/server request mixes can be visualised together. Architecture designs and their performance results can also be versioned within SoftArch/MTE and the performance results of these different architecture design options compared by architects.

6. High-level Software Architecture Modelling

SoftArch is comprised of a meta-model defining all available software architecture modeling abstractions (types), a model containing software architecture designs, system requirements (constraining architecture design information) and software designs (design-level classes refined from architecture abstractions), and a set of “design critics” that automatically assess designs and provide unobtrusive feedback to users. In addition a visual meta-model editor and visual model editing tools provide (mostly graphical) viewing and editing support [12].

We have developed a SoftArch meta-model to support encoding high-level, complex software architecture designs for the purpose of supporting the generation of MTE-style test bed code. The main elements of this meta-model are shown in our visual meta-model editor in Figure 5. Details of the element types and properties are shown in The key architectural modelling abstractions we have defined for SoftArch/MTE include clients, servers, server objects, database servers, and database tables. Each client or server object may have one or more requests to another object or to a database. A server object provides one or more remote services, basically a grouped set of server-side requests, for multi-tier architectures. These client element requests and server element services may be used at the architectural level to represent a number of grouped implementation-level methods. An architectural element may represent a high level abstraction e.g. “Clients” or a low-level one e.g. “RemoteVideoDataManager”. Architecture elements are linked by connectors of different kinds e.g. belongs-to, implements, data usage, method call, message passing, event subscribe/notify, hosted-by.

A number of properties for each of these architectural abstractions exist. These may be structural properties e.g. names, types, roles, arities and constraints over structure. These are denoted in Figure 6 with “AP”. Other properties are used by the performance test-bed code generator e.g. number of times to call a server request, number of concurrent threads to create for a client or server, pause duration (if any) between making requests, number of rows and columns a database table has, number of rows expected to be returned or updated by a database query, number of arguments a server request expects and so on. These are denoted by “CG” in Figure 6. Properties may have single values, multiple values or expressions. Structural properties express constraints over the architectural model and both structural and code generation properties are used by the SoftArch/MTE code generator in the production of test-beds code and scripts. Architectural abstractions may be successively refined by “refinement relationships” [12] e.g. a high-level “Servers” abstraction to multiple lower-level abstractions e.g. VideoDataManager, RentalDataManager, DatabaseServer and so on. Properties of these abstractions are also successively refined and consistency checking between supported [12].

A user of SoftArch/MTE is able to model architectures using these available abstractions which are understood by our test bed code generator. The architect parameterises software architecture designs with available element and connector properties. For example, they may specify a method call between client request and server service is implemented by CORBA and 100 calls are to be made, and have code generated to implement this in their test bed. They may then change this to RMI and 250 calls for generation of a new test bed and subsequent test run. The available abstractions can be changed or added to using the visual meta-model editor (e.g. to define a more specialised kind of architectural element or provide further characteristics about an element) but the test bed code generator scripts need to be modified if this is done.

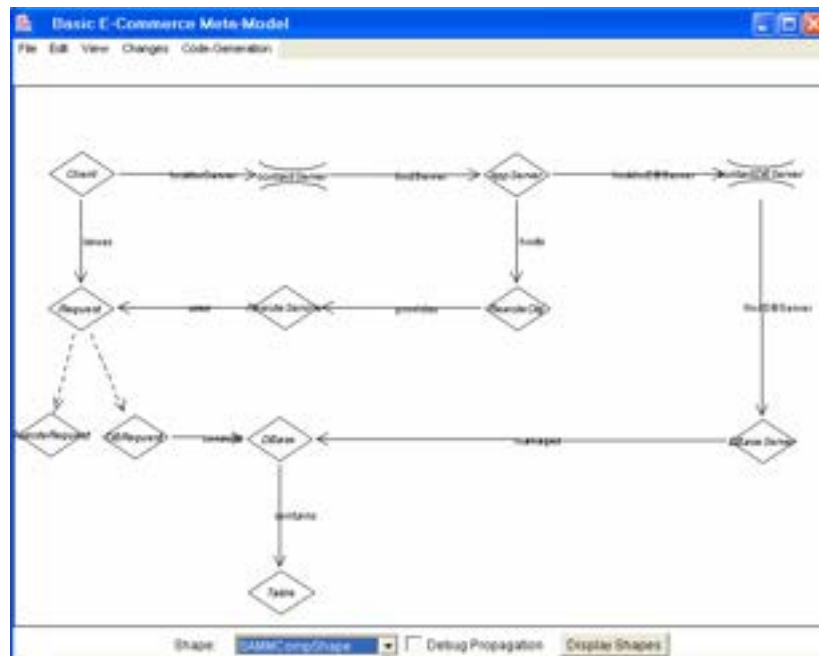


Figure 5. A visual representation of the SoftArch/MTE meta-model for E-commerce applications.

Element Type	Main Attributes	Property Description
Client	ClientType (AP, CG) Threads(CG)	Type of a client. For example, a client can be a browser, a CORBA client, or a RMI client. Number of clients that will be running during performance testing.
RemoteRequest	RemoteServer (AP, CG) RemoteObject(AP, CG) RemoteMethod(AP, CG) RecordTime(CG) TimesToCall(CG)	The name of remote server, to which the remote request is launched The name of remote object, which provides services required by the remote request The name of remote service, which completes the remote request A switch/boolean that defines if the performance testing results are recorded or not. Repetition time of certain operations during performance testing
DBRequest	QueryType (AP, CG) Dbase(AP, CG) Table(AP, CG) TimesToCall(CG) RowsReturned(CG) RecordTime(CG) Caching	Type of query for test bed and performance testing, such as 'select', 'update', 'insert', etc. The name of queried database The name of queried table Repetition times of certain operations during performance testing Number of returned results of the database request A switch/boolean that defines if the performance testing results are recorded or not. A switch/boolean that defines if query results are cached or not
AppServer	RemoteObjs (AP, CG) Type (AP, CG)	Names of all objects this application server hosts Type of the application server, such as CORBA, RMI, and J2EE.....
RemoteObject	Type (AP, CG)	Type of the remote object. Type could be CORBA, RMI, and EntityBean.....
RemoteService	Arguments (CG) Threading (CG) ConcurrencyControl (CG) RecordingTime (CG)	Arguments used by the service/operation A Boolean that records if this service symbols multi-threads character or single-thread character A Boolean parameter that records if the service has concurrency control or not A switch/boolean that defines if the performance testing results are recorded or not.
DBServer	Dbases (AP) ServerType (AP, CG)	Name of all databases this database server hosts Type of database server, such as MSSQL, Cloudscape, and ORACLE.....
DBase	Name (AP, CG) Type (AP, CG)	Database name Type of database server, such as MSSQL, Cloudscape, and ORACLE.....
Table	Name (AP, CG) PrimaryKey (CG) Rows (CG) Columns (CG)	Table name The primary key of the table Number of rows of data expected in table Number of columns expected in table

Figure 6. Some SoftArch/MTE meta-model types and properties.

Figure 7 shows an example 3-tier architecture for part of the on-line video system modelled using the SoftArch/MTE meta-model abstractions from Figure 5. In this example, staff and customer client programs have a number of requests that they can make on remote services hosted elsewhere e.g. find video/customer/rental, add/update rental item, update customer details etc. Some of these requests are simple (one remote call), others involve several remote requests. For each client, a number of "users" is specified and for each client request a number of times to call the remote service(s) and time to pause (if any) between invocations can be specified by the architect. This information configures the actual server loading tests that will be run by SoftArch/MTE's generated performance test-beds.

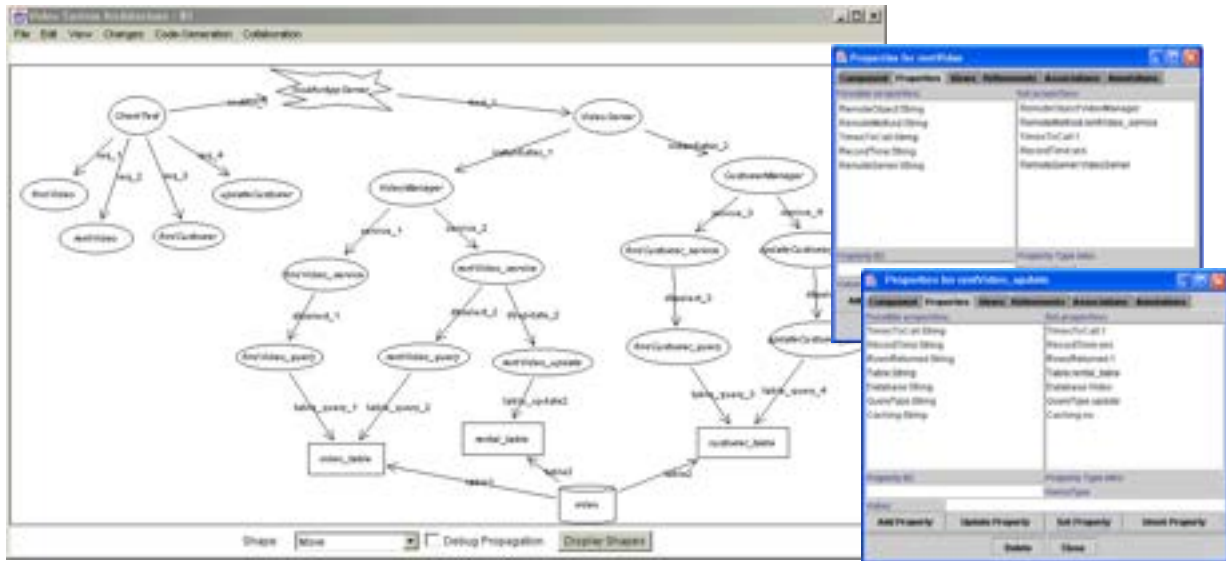


Figure 7. SoftArch/MTE meta-model abstractions and example high-level distributed system architecture.

In this example the customer clients connect to a set of remote objects (VideoManager, CustomerManager etc). These could be EJB objects/servers, CORBA or COM objects, CORBA server or DCOM server processes, Java Server Pages (JSPs), servlets or ASPs and so on. We use remote CORBA objects in most of our examples in this paper. In this architecture design, two remote objects are hosted by a single CORBA application. Similarly, for this architecture design a single database stores data (customers, staff, videos, rentals, etc). Connections between architectural elements specify request/service ownership, client-server-server connectivity and so on.

Various properties of architecture elements and connectors are specified via dialogues. These include number and kind of each request expected; kind of remote service, remote service requests, database table properties (expected number of rows/columns), client and server process hosts, and so on. Multiple views can be created by architects to enable specification of complex architectures or to provide various ways of viewing the same architecture design to help manage complexity. Figure 8 shows some of the properties for some elements defined for this architecture. Elements with a '*' have their properties shown at the right. Four different clients have been defined whose server requests will be run concurrently by the performance test bed. A single server defines two remote server objects. Each server object provides some remote services that are invoked by the clients across a CORBA middleware infrastructure. In this architecture (simple 3-tier), the server-side services run database transactions to select, insert and update data.

Meta-Type	Element	Sample Attributes&Values (for Element marked '*')
Client	ClientTest *	ClientType: CORBA Threads: 15
RemoteRequest	findVideo rentVideo * findCustomer updateCustomer	RemoteServer :VideoServer RemoteObject: VideoManager RemoteMethod: rentVideo_service RecordTime: yes TimesToCall: 1
DBRequest	findVideo_query rentVideo_query rentVideo_update * findCustomer_query updateCustomer_query	QueryType :update Dbase: video Table: rental_table TimesToCall: 1 RowsReturned: 1 RecordTime: yes Caching: no
AppServer	VideoServer *	RemoteObjs: VideoManager, CustomerManager Type:CORBA
RemoteObj	VideoManager * CustomerManager	Type: CORBA

RemoteService	findVideo_service rentVideo_service * findCustomer_service updateCustomer_service	Arguments: 1000 Threading: false ConcurrencyControl: true RecordingTime: yes
Dbase	Video *	Type:MSSQL
Table	Video_table * Rental_table Customer_table	PrimaryKey: ID Rows: 2500 Columns: 12

Figure 8. Some element types and their property values for the simple 3-tier video system architecture.

This video system could be realised by a wide variety of different architectures and middleware and database technologies to that shown in Figure 7. Each different architecture and middleware choice has advantages and disadvantages. The architecture from Figure 7 provides a single point of failure and the danger of bottlenecking the remote object services and database server. In addition, different clients, client requests and server services may suit a different architectural arrangement. For example, simple high-volume read requests may be cached or split across multiple machines with mirrored databases, whereas different kinds of transaction processing updates are centralised on one or more machines according to their data access or remote object service access patterns [1, 10, 35]. Architects may want to use a different set of architectural design choices and compare their estimated performance metrics.

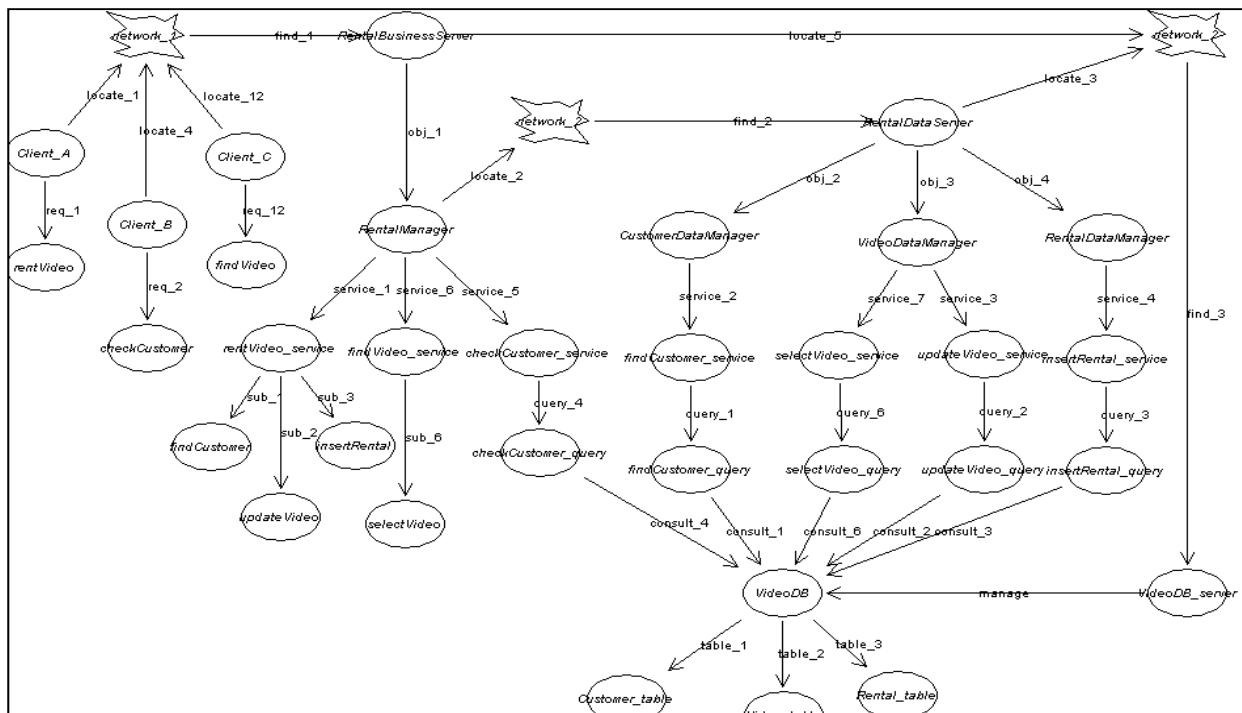


Figure 9. An alternative video system architecture design.

An alternative architecture design for our video system modelled in SoftArch/MTE is shown in Figure 9. In this design the architect has used a combination 3-tier and 4-tier architecture. Two CORBA servers host a RentalManager (2nd tier) and RentalDataManager (3rd tier) functionality. The RentalManager handles rental processing logic and customer database updates, while the RentalDataManager handles all data updates associated with rental processing. The motivation for this change is that the architect wants to discover if this splitting of server load will improve performance based on the expected client-request mix. The architect has specified different server-side services to invoke for each of the client

requests, along with a different number of requests for each and differing delays between them. Some clients have a large number of concurrent threads (simulating a larger pool of likely users) while others a small number. The architecture elements, their types and some sample properties for this alternative architecture are shown in Figure 10.

Meta-Type	Element	Sample Attributes&Values (marked with ‘*’)
Client	Client_A* Client_B Client_C	ClientType: CORBA Threads: 15
RemoteRequest	RentVideo * checkCustomer findVideo	RemoteServer :RentalBusinessServer RemoteObject: RentalManager RemoteMethod: rentVideo_service RecordTime: yes TimesToCall: 10
DBRequest	checkCustomer_query findCustomer_query selectVideo_query updateVideo_query * insertRental_query	QueryType :update Dbase: video Table: rental_table TimesToCall: 1 RowsReturned: 1 Caching: no RecordTime: yes
AppServer	RentalBusinessServer * RentalDataServer	RemoteObjs: RentalManager Type:CORBA
RemoteObj	RentalManager CustomerDataManager VideoDataManager * RentalDataManager	Type: CORBA
RemoteService	rentVideo_service* findVideo_service checkCustomer_service findCustomer_service selectVideo_service updateVideo_service insertTental_service	Arguments: 1000 Threading: false ConcurrencyControl: true RecordingTime: yes

Figure 10. Elements, types and example properties for complex video architecture.

7. Generating Performance Test-bed Code

From high-level architecture designs SoftArch/MTE can generate test beds that will provide realistic performance measures for a developed system based on the architecture designs specified. Note that the degree of accuracy will depend on the amount of information the architect specifies and the eventual accuracy of this information. The more precisely the architect identifies likely client->server requests and database updates and the closer the mix of specified requests on servers and databases are the more accurate will the performance test-bed results obtained. We have discovered that architects will usually over-specify requests and database access/update number/frequency to get performance results at the “lower end” of the performance range of an eventual system i.e. will try and get pessimistic rather than optimistic performance results. During architecture design and system development the architect can always update their architecture design in SoftArch/MTE, re-run performance tests and see how the likely system performance may change. Our experiments to date with SoftArch/MTE performance test-bed results and comparison to implemented system performance have shown useful performance estimates are usually achieved from even a very small time investment in high-level architecture design.

The test-bed code generated by SoftArch/MTE is currently “deterministic” i.e. the specified number and ordering of client/server requests, number of threads, pauses between requests, and deployment of clients and servers onto hosts is fixed. The advantage of this is that architects can change parameters associated with architectural elements and the effects

of these changes on performance results are reflected by the changes made to the generated test-bed code. In addition, the more “accurate” an architect wants the test-bed results to be, the more detailed a mix of client and server requests they can specify. The additional test-bed code that is generated will have finer-grained client/server/database code, resulting in both more fine-grained architecture element communication and more detailed performance analysis results for the architect. The disadvantage is that the architect must specify all parameter values for number of users/threads, number of transactions to try, time limit for results capture and so on. They may, of course, get these wildly inaccurate, and will the generated test-bed will capture performance results for this hand-crafted set of interactions. It would, however, be possible with our approach to generate test bed code that created random (within architect-specified ranges) number of threads/users, pauses between requests, number of client requests and so on.

The code generation process used by SoftArch/MTE is outlined in Figure 11. SoftArch/MTE converts the architecture design built by its user into an XML encoding of this design (1). A collection of XSLT transformation scripts and an XSLT engine (2) is used to generate the performance test-bed client and server program source code, IDLs, deployment descriptors, compilation scripts, deployment scripts, database table construction scripts, and so on (3). Client and server program code is compiled automatically by SoftArch/MTE using generated compilation scripts (4) to produce fully-functioning, deployable test-bed code (5).

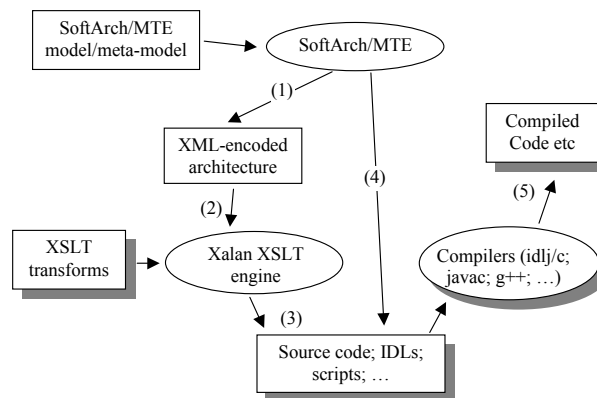


Figure 11. Code generation process.

We chose to use an XML encoding of our SoftArch architecture designs as this provides a design model that can be readily exchanged between tools and be processed by XML-oriented tools. It was natural thus to use a set of XSLT transformation scripts to implement our code generation. The key advantages of adopting XSLT-based code generation from XML-encoded architecture designs include:

- XSLT transformations are easily modified and extended, or new scripts for new kinds of source code, compilation script, IDL etc generation added, without the need for complex coding or modifying SoftArch/MTE source
- de-coupling of our code generation and architecture modelling and design representation
- allows new meta-model abstractions to be added using the SoftArch meta-modelling tool that will add new XML items to our architecture representation without breaking the existing code generation;
- allows other CASE tools to import the XML-encoded architecture designs at the beginning full system design/code generation
- will possibly allow other architecture design tools to generate the XML architecture encoding but use our code generation XSLT scripts.

Figure 12 shows part of the architecture model (client request to server remote service) plus examples of XML-encoded architecture design information generated by SoftArch/MTE representing this information. (b) shows the client specification and (c) shows a server remote object specification. The XML represents the architecture elements, their relationships and structural and code generation properties. We found XSLT scripts applied to these XML architecture models provided us with a powerful code generation facility that was significantly easier and quicker to use than using standard programming languages to generate code (as we used in previous CASE tool projects [14]).



Figure 12. (a) Part of architecture model; (b) XML encoding of some of model.

To give an idea of the way we used XSLT scripts to generate performance test-bed code, Figure 13 shows part of an XSLT transformation script used to convert parts of the XML matching CORBA client requests into CORBA client code in Java.

<pre> <?xml version="1.0"?> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"> <xsl:output method="xml" indent="yes"/> <!-- <xt:document href="{Name}.java" --> <!-- generate class header & constructor --> <xsl:template match="Client"> <MyJava> public class <xsl:value-of select="Name"/> { public static void main(String[] args) { int threadNumber=<xsl:value-of select="Threads"/>; int index=0; while(index!=threadNumber) { (new <xsl:value-of select="Name"/>_Thread(args)).start(); index++; } } } class <xsl:value-of select="Name"/>_Thread extends Thread { <xsl:for-each select=" RemoteRequest /RemoteObject"> private <xsl:value-of select="."/><xsl:text> </xsl:text> <xsl:value-of select="."/>_ SINGLETON; </xsl:for-each> public <xsl:value-of select="Name"/>_ Thread(String[] args) { <xsl:for-each select=" /Client/RemoteRequest"> if((<xsl:value-of select="RemoteObject"/>_ SINGLETON) == null) <xsl:apply-templates select="RemoteObject" mode="connect"/> </xsl:for-each> } public void run() { <xsl:value-of select="RemoteRequest/Name"/>(); } <xsl:apply-templates select="RemoteRequest"/> } </MyJava> </xsl:template> </pre>	<pre> <xsl:template match="RemoteObject" mode="connect"> try{ ORB orb = org.omg.CORBA.ORB.init(args, null); org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService"); NamingContext ncRef = NamingContextHelper.narrow(objRef); NameComponent nc = new NameComponent("<xsl:value-of select="."/>", " "); NameComponent path[] = {nc}; <xsl:value-of select="."/>_ SINGLETON = <xsl:value-of select="."/RemoteServer"/>Lib. <xsl:value-of select="."/> Helper.narrow(ncRef.resolve(path)); } catch(Exception e) { System.out.println("ERROR : " + e); e.printStackTrace(System.out); } </xsl:template> <xsl:template match="RemoteRequest"> public void <xsl:value-of select="Name"/>() { int iter = <xsl:value-of select="TimesToCall"/>; String recordTime = "<xsl:value-of select="RecordTime"/>"; System.gc(); long start = System.currentTimeMillis(); int i=0; while(i != iter) { <xsl:value-of select=" RemoteObject"/>_ SINGLETON. <xsl:value-of select=" ./RemoteMethod"/>(); i++; } if(recordTime.equals("yes")) { long time = System.currentTimeMillis() - start; double elapse = (double) (time) / (double) (Math.max(1,iter)); String perf = "<xsl:value-of select="//Name" /> _V \t"+time+"\t"+iter+"\t"+elapse; System.out.println(perf); //System.err.println(perf); } } </xsl:template> </xsl:stylesheet> </pre>
--	---

Figure 13. XSLT threaded client code generation script example.

The XSLT script contains Java code fragments that are “fixed” for all clients and XSLT instructions that copy values out of the XML architecture design encoding and generate Java function names, argument values, control construct expression values and so on. Each of the `<xsl:value-of select=“...”/>>` constructs is used to extract model element values to parameterise the code generation being done by the script. When each of the XSLT scripts is run for a given input file it substitutes the specified values in the XML file, referred to in the XSLT script by XPath expressions, for the actual architecture design values. This results in Java code files, batch scripts, deployment descriptors and so on being generated specific to the architecture design values given in the XML design encoding.

Figure 14 shows an example of a Java code file implementing a client application program for the performance test-bed for the part of the architecture design and XML client data encoding shown. The relationship between architecture elements in the XML encoding and the resultant client Java program fragment is illustrated. Various parameters specified as properties of the architecture design elements in the XML encoding have been translated into Java code fragment values, statements and classes by the XSLT-based code generation script above. This client can be compiled and run and will perform using multi-threaded clients the specified number of server requests, collecting and outputting the total time taken for these requests for further analysis.

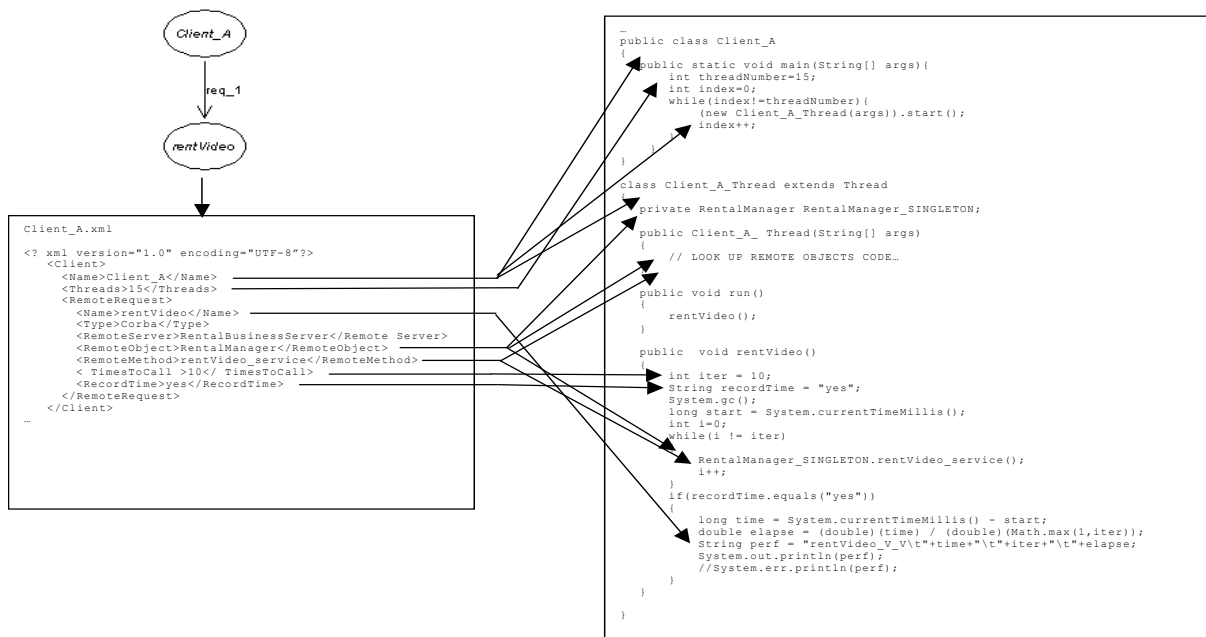


Figure 14. (a) Model XML and (b) example generated Java client code.

Different XSLT transformation scripts can match the same XML-encoded architecture data, generating different code. For example, `corba-client.xml` generates server object look-up functions for encoded CORBA server objects, whereas `corba-server.xml` generates object creation and registration code for this same XML-encoded data). These XSLT transformation scripts can be straightforwardly modified or new scripts added without requiring any SoftArch code or its XML encoding modification. We have found that this makes extending the code generation facilities of SoftArch/MTE much easier than language-implemented code generation. At present we have code generation scripts that produce CORBA, RMI, J2EE (Enterprise JavaBeans), Java Server Pages, Active Server Pages, ADO.NET and SOAP web service middleware performance test-bed code. All are implemented by different XSLT transformation scripts that are invoked by SoftArch/MTE on generated XML encodings of architecture designs.

8. Testing and Visualising Architecture Performance

Once performance test-bed code and script files have been generated the test-beds must be deployed, tests run and results captured and visualised. Our approach with SoftArch/MTE has been to try and fully-automate this testing and results capture process. A set of available test hosts are identified to SoftArch/MTE by architects – these provide the

machines on which the tests will be run. Our aim has been to focus on providing a realistic deployment scenario where generated client and server and database test-beds can be deployed realistically and thus test results obtained be as accurate as the architecture description and generated test-bed code permits. We developed a “deployment agent” which runs on all available test-bed program hosts and is used to upload information about the host to SoftArch/MTE and to download SoftArch/MTE-generated test bed code and scripts, deploy the test-bed code using the scripts, co-ordinate the running of tests on the host and upload results back to SoftArch/MTE.

Figure 15 outlines our test-bed code performance testing process. Generated code is compiled by SoftArch, using generated compilation scripts (1). The compiled code/IDLs/deployment descriptors etc and the scripts to configure and deploy these on a host are up-loaded to remote client and server hosts using the remote SoftArch/MTE deployment agents (2). The deployment agents organise the uploaded code and associated scripts into suitable directory structures, run the scripts to properly configure the host machine, its database and registries, and to deploy the test-bed code.

The client and server programs are then run: CORBA, RMI and other server programs are started; EJB, JSP and ASP components are deployed into J2EE and IIS servers; database servers started and database table initialisation scripts run; and finally clients are started (3). Clients look up their servers and then await SoftArch sending a signal (via their deployment agent) to run, or may start execution at a specified time. Clients send servers requests, logging performance timing for different requests to a file (4). Servers like-wise log the time taken to execute their remote methods and database operations. Performance results are currently collected in comma-separated value text files. These results are sent back to SoftArch/MTE after tests have completed (5).

SoftArch/MTE collects the test results and aggregates them using simple data processing algorithms to form a unified result set for the performance tests. The results are associated with SoftArch/MTE architecture model instances using data annotation facilities built into the SoftArch modelling tool’s repository. SoftArch/MTE then uses data visualisation techniques to display the results of the performance tests to architects, possibly using 3rd party tools like MS Excel™ (6).

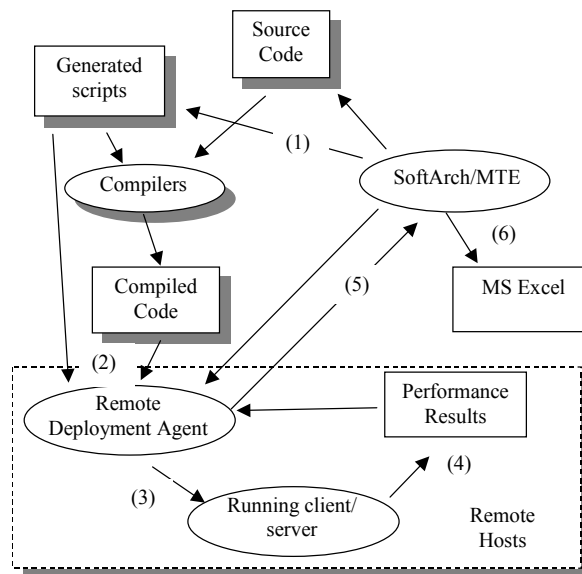


Figure 15. System deployment and test run process.

For thick-client applications the structure of the client-server relationships is straightforward, as illustrated in Figure 16. The client program is started (1) and it looks up the server object(s) it requires (2) which depending on the architecture design may be located on different remote hosts. When instructed by its deployment agent (3) the client runs its specified server remote request operations (4), collects the results of these (5) and the deployment agent uploads the results to SoftArch/MTE (6).

For thin-client applications, such as JSP and ASP-implemented server pages, SoftArch/MTE generates server-side implementations of these using (currently) JSPs or ASPs and these are hosted by a J2EE server or IIS server respectively (1). A “pseudo-browser” client running on client hosts is configured by scripts generated by SoftArch/MTE and run by our deployment tool (2). When instructed to perform its tests the pseudo-browser instructs its IE6 object (3) to request

page(s) from the servers (4), which may in term make database or remote object requests. The pseudo-browser client generates performance results to a file (5) and this data is uploaded to SoftArch/MTE by the deployment tool (6).

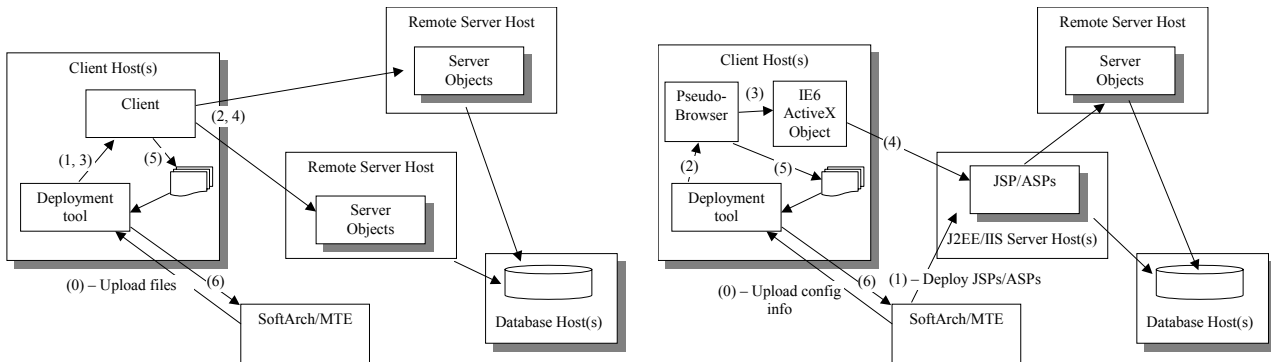


Figure 16. (a) Thick-client test-bed structure; and (b) thin-client test-bed structure.

The performance test-bed results currently include name of request/service, name of owning object/process/program, number of times called, overall time taken, and data stored/retrieved. When collecting this data SoftArch/MTE records performance results against appropriate architecture client requests, services, client/server objects and program elements. Data is summarised across all test bed client and server instantiations to produce an aggregate test result. Summarised results include number of calls made by a request/to a service; average and total time to complete request/service; average and total time spent in a request/service; and average and total database accesses/updates performed.

Architects need these performance results visualised so that they can determine the overall performance of an architecture, view (where possible) the performance of parts of the architecture e.g. time taken in one server method vs. another vs. in database; and ultimately can compare performance results for different middleware and database selections and ultimately for different architecture designs. In previous work on the MTE project we used graphs to describe hand-crafted performance test-bed results [10]. On the SoftArch project we provided a visualisation mechanism that allowed developers to visualise performance metrics in high-level SoftArch architecture diagrams by the use of annotations [12]. We wanted to reuse these two approaches in order to provide architects with high-level views of performance results obtained from running generated test-beds.

We reused the visualisation techniques described in [12], which were originally developed for real, fully-implemented architecture visualisation, to automatically visualise our generated test-bed results. The architect requests particular performance results they wish to view and a range of architecture design element annotations are used to visually indicate light vs. heavy requesting/loading, small vs. large time consumption, small vs. large data transferral, etc. An example of such a performance visualisation is shown in Figure 17 (1). This example shows an annotated form of the video system diagram from Figure 7 after a performance test has been run. In this example, the annotations are showing total time taken for different architectural elements and their interconnecting relationship. Note that the performance results obtained from client and server objects have been summarised by SoftArch/MTE to obtain aggregate data which is what the visualisation is using to construct its annotations. We wrote a “visualisation agent” for SoftArch/MTE that is plugged into SoftArch and converts the performance results collected by from our generated performance test-beds into the same form generated by our code annotations as described in [12]. This converted data is then used by our existing visualisation algorithm to generate the annotations illustrated in Figure 17 (1).

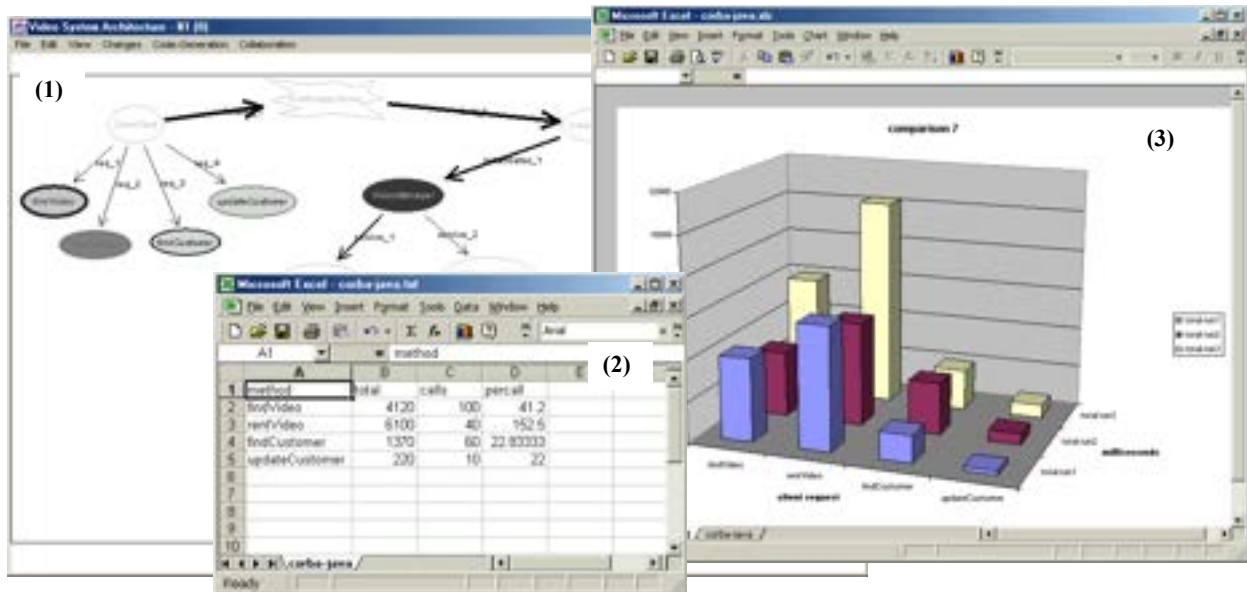


Figure 17. Visualising test run results in SoftArch/MTE and MS Excel™.

SoftArch/MTE performance results are summarised into comma-separated value files by deployment agents. SoftArch/MTE further aggregates results from all deployment agents hosting generated client and server test-bed programs. Figure 17 (2) shows an example of performance result values for one client collected by SoftArch/MTE and opened for display using a MS Excel™ spreadsheet. When results are aggregated into a single result set across all clients and servers collecting performance figures these can be visualised in SoftArch/MTE or in other tools, such as MS Excel graphs. Figure 17 (3) shows aggregated results from three separate performance tests of the video system. Each of these uses a different architecture design for this example system - multiple CORBA objects in one server process; remote objects split across 2 different server processes; and remote objects split across 3 server processes. SoftArch/MTE aggregated the results of generated architecture performance tests into one Excel file and then an MS Excel™ chart was used to chart these values for comparison by the architect.

9. Discussion

9.1. Experiences with SoftArch/MTE

During our research work we have used SoftArch/MTE to help us evaluate the performance of several simple architectural designs for the video system. These designs involved different organisations of server-side functions e.g. single remote object with all functions; multiple remote objects each with different functions; multiple objects deployed on different hosts; and multiple instances of the same object on different hosts. They also involved centralisation of all database information and splitting of database information across multiple database servers and database server hosts; specification of caching for some data; threading and synchronisation choices for remote objects and their services (methods), and choice of different middleware (EJBs, CORBA and RMI objects) and database servers (SQL Server, Cloudscape and Interbase). We also generated thick-client applications (Java applications using remote CORBA, RMI and/or Enterprise JavaBean objects) and thin-client applications (using Java Server Pages and Active Server Pages). The thin-client applications have JSPs or ASPs hosted by web servers (Apache and IIS respectively) and have JavaBeans and .NET components generated to encapsulate data and remote object/database connectivity code.

The SoftArch meta-model and the architecture descriptions for these different client, middleware and database specifications are very similar with subtypes and type properties used by the architect to select different choices. Architecture designs are different assemblies by the architect of client, server, remote object, database, request, service and database table abstractions. For each kind of client, remote server object and middleware/database configuration we wrote XSLT scripts that traverse the XML-encoded SoftArch/MTE architecture specification. These scripts can be easily modified to generate differently-structured code. For example, we changed our thin-client generation scripts from generating stand-alone JSPs and ASPs to generating JSPs that use related JavaBean classes and ASPs that use .NET web

components. This was to produce web server page implementations using more realistic design and implementation structures for testing purposes. The overhead of changing the XSLT scripts was very low with no changes needed to SoftArch/MTE or its XML model to generate these much more complex, fine-grained web server page test-bed implementations.

The main technical challenges we faced when building SoftArch/MTE include identifying and defining suitable architecture abstractions and properties; using XML encodings of architectures to generate suitable test-bed code; capturing appropriate performance measurements from the generated test-bed code; and providing useful visualisations of these measures to architects. SoftArch/MTE allows architects to model a range of high-level and lower-level architectural designs. Our experience with the tool suggests that when an architect can't model the architectural abstractions they want, it is usually due to an incompleteness of the meta-model and its associated code generation scripts. Currently it is not difficult for architects to extend these, but it can be very time-consuming and error-prone. More work is needed both to improve the modelling language of SoftArch/MTE to make it easier to model architectural abstractions and to improve its customisability. Currently only simple generated code instrumentation is used to collect performance results and simple aggregation and visualisation of these results is supported.

8.2. Performance Results

To try and get an idea of the potential accuracy of our SoftArch/MTE performance test-bed generator results, we compared the performance of the video system 3-tier architecture test bed completely generated by SoftArch/MTE to the performance results obtained when running the same client performance tests on a fully-implemented version of the system's servers built as part of a previous teaching project. For this test we used a thick-client (Java application), CORBA middleware and SQL Server 8 database. Several services were modelled, including simple data access (e.g. findCustomer) and simple data updates (updateVideo) that in the real system are passed object parameters and perform simple business logic constraint checking and single database manipulations. Complex data querying (findVideos) and services with complex business logic processing and data update (rentVideo and returnVideo) were also modelled, that in the real system perform multiple database queries and updates. Two sets of results are summarised in Figure 18. One shows averaged results across 5 test runs with 100 requests of each type to the server, using a single-threaded client. The second shows averaged results across 5 test runs for 10 of the same requests when 10 concurrently running clients are used (where other server threads run during network and database blocking). In each case, the same generated test-bed clients were run on both the generated test-bed servers and the hand-implemented servers (with some small hand-modifications to the generated clients to provide appropriate required parameter values for the real servers).

Single-threaded client tests				
	Generated Server (ms)	Actual System Server (ms)	Difference (ms)	% diff
rentVideo	684.4	909.6	225.2	24.75
returnVideo	459.4	640.4	181	28.26
findVideos	227.8	312.6	84.8	27.12
findCustomer	181.6	262.6	81	30.84
updateVideo	252.8	256.4	3.6	1.40
Multi-threaded client tests (10 concurrent clients)				
	Generated Server (ms)	Actual System Server (ms)	Difference (ms)	% diff
rentVideo	435.28	631.42	196.14	31.06
returnVideo	433.48	603.6	170.12	28.18
findVideos	157.18	220.64	63.46	28.76
findCustomer	154.26	202.08	47.82	23.66
updateVideo	202.86	225.84	22.98	10.17

Figure 18. Results of generated test-bed server code performance vs. implemented server performance.

Client request	Actual system performance result	Generated System performance result	Difference (ms)
productManagePage	27ms	26ms	-1
accountManagePage	23ms	24ms	+1
orderManagePage	39ms	28ms	-11

Figure 19. Results of C#.NET-implemented Pet Shop application tests.

We also modelled the architecture of the Pet Shop reference architecture and compared a C#.NET version (a thin-client architecture) to a freely-available 3rd party implementation of this system developed by Microsoft Corp. using C# and ASP.NET. In this system, the servers include web page content construction (via ASPs) as well as database access, update and simple business logic. The clients were run with 8 concurrent threads and no pause between requests to the servers. Results are shown in Figure 19. Again we compared the results obtained running the same SoftArch/MTE client performance testing code on both generated test-bed server code and existing system hand-implemented server-side code. For the real Pet Shop system we again had to add by hand some example URL argument data to the ASP server page calls from the clients to ensure they worked correctly.

In the above tests, the performance of the generated test-beds to the hand-implemented systems was generally quite close. When a small number of server-to-server or server-to-database requests are made by the server-side abstractions, the results are generally very close, as in the Pet Shop example. With larger number of complex server-side services, such as the video system's rent/return video processing, the results can vary due to business logic in the real system which is not present in the generated test-beds, and due to additional synchronisation code. For example, the generated orderManagePage service in the Pet Shop example above lacks some synchronisation code that is in the real system, resulting in the test-bed performance being artificially very high. In addition, in both examples the test bed does not pass complex datasets between requesting objects and service. The performance impact of this is noticeable in the findVideos and findCustomer examples for the video system, where these services have multiple arguments in the real implementation.

We have also used SoftArch/MTE to help us model some other system architectures and compare the performance of different architecture and middleware design choices. We designed architectures for a micro-payment system [6] that we have implemented as part of another research project which uses CORBA, J2EE JSPs and EJBs and relational databases. We modelled an on-line travel planning system used on previous research and teaching projects: one using a peer-to-peer decentralised RMI architecture, one using Java Server Pages and JDBC, one using Active Server Pages and ADO.NET, and a final variant implemented using C# and .NET SOAP web services. We have also designed simple SoftArch/MTE architectures for an enterprise workflow system and an integrated health informatics system (which provides a range of heterogeneous clients and servers using EJBs, XML and CORBA). With each of these systems we modelled a number of remote server interfaces using SoftArch/MTE and a mix of thick- and thin-client user interfaces. In most cases we modelled simplified forms of the architectures of the fully-implemented systems, with fewer client requests and server services than in the target systems.

In general, the anecdotal performance results obtained when running the fully-generated test-beds for most simple client requests and data-oriented server processing functions have been of a similar degree of closeness as the above examples, to those obtained by running the same client test programs against the manually-implemented servers. However, complex client requests and server services with program logic we can't currently model with SoftArch/MTE e.g. complex algorithms and decision logic affecting the number of subsequent remote object calls and database updates. These can thus provide very inaccurate results. In addition, SoftArch/MTE provides abstractions describing client/server/database request interactions an architect can only use SoftArch/MTE to model an estimated number of remote requests and data updates/accesses for such services. These may well be very inaccurate and be very different in the final system implementation. However, SoftArch/MTE can still be used to obtain performance information incrementally during development with the architect refining the estimated number of flow-on requests for a service to produce more useful performance estimates.

8.3. Evaluation of Our Approach

Key advantages of the SoftArch./MTE approach to performance test-bed generation include the ease with which architects can specify an architecture design and have realistic performance results obtained; the extensibility of our code generation and meta-modelling approach to supporting this performance test-bed generation; and the use of real code and

machines to run the test-beds, thus obtaining “realistic” results. Our approach allows results like those obtained from hand-coded architecture and middleware performance analysis projects [10, 17, 27] to be obtained with much less effort on the part of developers. In addition, a major advantage is that developers can change their architecture designs and middleware choices, having their performance test-beds completely automatically regenerated and performance tests re-run. We have incrementally enhanced our toolset by adding new meta-model abstractions and code generation support (e.g. for modelling and generating thin-client applications using JSPs and ASPs) without impacting on existing architecture models and SoftArch architecture tool implementation. XSLT has proved to be a very flexible, powerful tool for implementing our extensible code generation facility from XML-encoded architecture models.

Our current approach to performance test-bed generation has some key limitations. The architect is constrained to use the provided meta-model abstractions and XSLT code generation scripts. In order to use other middleware, they or others must extend the SoftArch/MTE meta-model and its code generation scripts, the latter essentially programmatic effort. This work can be challenging e.g. it took us several weeks to build working EJB code generation and deployment scripts and around three person-months to build basic JSP and ASP thin-client generation scripts. As it is often hard for an architect to estimate the likely mix of client requests, database access and update requests and server-to-server requests in a final system when doing architecture design, such inaccurate loading specifications will naturally produce inaccurate performance measures. We had this experience when specifying architectures for our travel planning system and forgetting to account for some common client requests and server processing scenarios. This resulted in very unrealistic performance results being obtained compared to running the same client tests on parts of the fully-implemented system.

However, it must be emphasised that **all** testing approaches (prototypes, simulation and even monitoring fully developed systems) suffer from this same problem when using estimated loading rather than using real users and real data. Thus any of these approaches to estimating system performance based on a certain architecture design is only that - an estimate. We have noticed the ease in specifying discrete client and server requests in SoftArch/MTE architecture designs does usually lead to architects, especially novices who have tried using the system, specifying overly-simple designs. However, we suggest tools like SoftArch/MTE not be used as a once-off architecture performance test-bed generator but have an evolving architecture design kept up-to-date with development work on a project. The architect needs to continually refine their model and more fine-grained client-server requests be specified as the architecture design evolves, re-running performance tests periodically to track changing likely eventual-system performance hot-spots.

An additional requirement of our approach is the need for a reasonable number of client and server host machines to be available to run test-beds on in order to obtain meaningful results. If only a small number of client and server hosts are available for testing, SoftArch/MTE can not directly estimate likely performance on larger numbers of machines due to the need to co-locate some client and/or server test-bed code that in reality would not be run on the same machine or use the same kind of inter-machine network. In such scenarios, an architecture performance simulation technique may be able to produce more accurate performance estimations by modelling large numbers of heterogeneous client and server hosts that are not available for a SoftArch/MTE user.

8.4. Future Research

Currently SoftArch/MTE generates a performance test-bed using fixed values for architecture structural and code generation properties. The performance results obtained are fixed values of number of transactions performed or time taken for the transactions to complete. A possible extension would allow architects to specify ranges of values e.g. 9-12 concurrent threads; repeat request 4-6 times; and so on. SoftArch/MTE could then run the tests with different values for parameters and report the results as value ranges. Alternatively it could select random values within the ranges and report results, possibly after running the tests several times. These approaches would provide software architects with ranges of performance results, including maximums, minimums and standard deviations. These may be more useful for architecture design performance analysis than the current fixed-value results reported at present.

We have been adding new code generation scripts to SoftArch/MTE and meta-model abstractions to support generating message-oriented middleware (e.g. Java Messaging Service and MQ Series) code and are looking to generate code for SOAP (web services) middleware, both for Java and .NET platforms. We have implemented some basic C# code generation scripts in order to performance test architecture designs with this intended target implementation language. We are working on improved, richer performance result visualisations, including showing results for different test runs on (slightly) modified, overlapping architectures together. Currently SoftArch/MTE uses its own architectural representation in XML. We are investigating using a “standard” XML-based architecture design encoding, like xADL [21] or the XMI UML encoding, allowing other tools to use our code generation and deployment scripts.

We have recently been extending the open-source UML design CASE tool Argo/UML [31] to provide a SoftArch-style modelling diagram and meta-modeller for these diagrams, with extensions to the XMI encoding used by Argo/UML

to represent these architecture designs. We have extended the UML's support for architecture modelling to enable the specification of SoftArch/MTE-style architecture designs with test-bed code generation properties. This tool provides users with a SoftArch/MTE-style performance test-bed generator within a more common CASE environment. We plan to test this environment with industry co-operation on some large enterprise system performance analysis and tuning projects. We are also planning to investigate the extraction and update of architecture designs within this extended Argo/UML environment from Argo's OOD designs. This will allow users to automatically update their architecture fine-grained client requests and server services from an evolving system design, running re-generated test beds to determine how a system with an evolving, partially-implemented architecture is likely to perform if implementation is completed.

10. Summary

We have developed a prototype performance test-bed generator that automates the generation of performance prototypes, running of prototypes and visualisation of test-bed prototype performance using high-level software architecture models. Our SoftArch/MTE tool provides a high-level, extensible architectural modelling language that is rich enough to allow fully-working test bed code to be generated. We encode the SoftArch/MTE architecture designs in XML and use a set of extensible XSLT transformations scripts to transform an XML-encoded architecture design into test bed client and server program code and compilation/deployment scripts. A deployment agent running on available client and server hosts is used to automatically upload compiled systems and to configure and deploy them. Test runs are performed and the performance results captured by client and server programs are automatically captured and aggregated by SoftArch/MTE. These results are visualised by either the annotation of in the high-level architecture design diagrams within SoftArch/MTE or by 3rd party applications like MS Excel™.

We have used SoftArch/MTE to model a number of distributed systems, generate performance test-beds for these models, capture results and compare these results to performance measures obtained from hand-implemented, completed distributed systems. Our experiences have demonstrated that our approach provides a useful, accurate automated architecture performance analysis environment for these distributed systems. Main limitations of our approach include the need for a number of client and server host machines in order to get a realistic distribution of client and server test-bed programs, and the need for architects to identify and specify the "key" client requests and database processing for the architecture in order for useful results to be obtained. Use of SoftArch/MTE within a development process where architecture designs are continually updated and test-beds re-generated and tests re-run allows architects to incrementally refine their architecture designs and remain aware of the likely performance of a completed system based on this design. A tool like SoftArch/MTE can thus be used as a key enabler of proactive architecture design and performance-directed architecture evolutionary development.

Acknowledgements

Parts of this work were supported by the University of Auckland Research Committee, the New Zealand Public Good Science Fund and the New Zealand New Economy Research Fund.

References

1. Balsamo, S., Simeoni, M., Bernado, M. Combining Stochastic Process Algebras and Queueing Networks for Software Architecture analysis, In *Proceedings of the 3rd International Workshop on Software and Performance*, Rome, Italy, July 214-26 2002, ACM Press.
2. Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice*, Addison-Wesley, 1998.
3. Beaumont, M. and Jackson, D. Visualising Complex Control Flow. In *1998 IEEE Symposium on Visual Languages*, Halifax, Canada, September 1998, IEEE.
4. Chen, M., Tang, M. and Wang, W. Software Architecture Analysis - A Case Study, In *Proceedings of COMPSAC'99*.
5. CSIRO, Middleware Technology Evaluation project, www.cmis.csiro.au/mte.
6. Dai, X. and Grundy, J.C. Architecture for a Component-based, Plug-in Micro-payment System, In *Proceedings of 5th Asia-Pacific Web Conference*, September, Xi'an, China, Springer LNCS.
7. ECPerf Performance Benchmarks, August 2002, ecperf.theserverside.com/ecperf.

8. Egyed, A. and Kruchten, P., Rose/Architect: a tool to visualize architecture, In *Hawaii International Conference on System Sciences*, Jan. 1999, IEEE.
9. Gomaa, H., Menascé, D., and Kerschberg, L. A Software Architectural Design Method for Large-Scale Distributed Information Systems, *Distributed Systems Engineering Journal*, Sept. 1996, IEE/BCS.
10. Gorton, I. And Liu, A. Evaluating Enterprise Java Bean Technology, In *Proceedings of Software - Methods and Tools*, Wollongong, Australia, Nov 6-9 2000, IEEE.
11. Grundy, J.C., Hosking, J.G. ViTABaL: A Visual Language Supporting Design by Tool Abstraction, In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, Darmstadt, Germany, September 1995, IEEE CS Press, pp. 53-60.
12. Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, In *2000 IEEE Symposium on Visual Languages*, IEEE.
13. Grundy, J.C. and Liu, A. Directions in Engineering Non-Functional Requirement Compliant Middleware Applications, In *Proceedings of the 3rd Australasian Workshop on Software and Systems Architectures*, Sydney, Australia, Nov 2000, Monash University.
14. Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *Information and Software Technology: Special Issue on Constructing Software Engineering Tools*, Vol. 42, No. 2, January 2000, pp. 117-128.
15. Grundy, J.C., Cai, Y. and Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, In *Proceedings of the 2001 IEEE International Conference on Automated Software Engineering*, San Diego, CA, Nov 26-29 2001.
16. Hill, T., Noble, J. Visualizing Implicit Structure in Java Object Graphs, In *Proceedings of SoftVis'99*, Sydney, Australia, Dec 5-6 1999.
17. Hu L., Gorton, I. A performance prototyping approach to designing concurrent software architectures, In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, IEEE, pp. 270 – 276.
18. Juiz, C., Puigjaner, R. Performance modelling of pools in soft real-time design architectures, *Simulation Practice & Theory*, vol.9, no.3-5, 15 April 2002, Elsevier, pp.215-40.
19. Jurie, M.R., Rozman, I., Nash, S. Java 2 distributed object middleware performance analysis and optimization, *SIGPLAN Notices* 35(8), Aug. 2000, ACM, pp.31-40.
20. Kazman, R. Tool support for architecture analysis and design, In *Proceedings of the Second International Workshop on Software Architectures*, ACM Press, 94-97.
21. Khare, R., Guntersdorfer, M., Oreizy, P., Medvivovic, N. and Taylor, R.N. xADL: Enabling Architecture-Centric Tool Integration With XML, in *Proceedings of the 34th Hawaii International Conference on System Sciences*, Jan 3-6 2001, Maui, Hawaii, IEEE.
22. Lee, S.C., Offutt, J. Generating test cases for XML-based Web component interactions using mutation analysis, In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Hong Kong, China, 27-30 Nov 2001, IEEE CS Press.
23. Leo, J. OO Enterprise Architecture approach using UML, In *Proceedings of the 2nd Australasian Workshop on Software Architectures*, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
24. Liu, A. Dynamic Distributed Software Architecture Design with PARSE-DAT, In *Proceedings of Software – Methods and Tools*, Wollongong, Australia, Nov. 2000, IEEE.
25. Luckham, D.C., Augustin, L.M., Kenney, J.J., Veera, J., Bryan, D. and Mann, W. Specification and analysis of system architecture using Rapide, *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April 1995, 336-355.
26. Ma, Y-S. Oh, S-U. Bae, D-H., Kwon, K-R. Framework for third party testing of component software. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference*, IEEE CS Press, 2001, pp.431-434
27. McCann, J.A., Manning, K.J. Tool to evaluate performance in distributed heterogeneous processing. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, IEEE, 1998, pp.180-185.
28. Microsoft Corp., Using .NET to implement Sun Microsystem's Java Pet Store J2EE BluePrint application, October 2002, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/psimp.asp>.

29. Nimmagadda, S., Liyanaarachchi, C., Gopinath, A., Niehaus, D. and Kaushal, A. Performance patterns: automated scenario based ORB performance evaluation, In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, USENIX, 1999, pp.15-28.
30. Petriu, D., Amer, H., Majumdar, S., Abdull-Fatah, I. Using analytic models for predicting middleware performance. In *Proceedings of the Second International Workshop on Software and Performance*, ACM 2000, pp.189-94.
31. Robbins, J. Hilbert, D.M. and Redmiles, D.F. Extending design environments to software architecture design, *Automated Software Engineering*, vol. 5, No. 3, July 1998, 261-390.
32. Shannon, B., *Java 2 platform, enterprise edition : platform and component specifications*, Addison-Wesley, 2000.
33. Shaw, M. and Garlan, D. *Software Architecture*, Prentice Hall, 1996.
34. Topol, B., Stasko, J. and Sunderam, V., PVaniM: A Tool for Visualization in Network Computing Environments, *Concurrency: Practice & Experience*, Vol. 10, No. 14, 1998, pp. 1197-1222.
35. Vogal, A. CORBA and Enterprise Java Beans-based Electronic Commerce, *International Workshop on Component-based Electronic Commerce*, Fisher Center for Management & IT, UC Berkeley, 25th July, 1998.
36. Woodside, C. Software Resource Architecture and Performance Evaluation of Software Architectures, In *Proceedings of the 34th Hawaii International Conference on System Sciences*, IEEE, Maui, HA, Jan 2001.

4.4 Realistic Load Testing of Web Applications

Draheim, D., Grundy, J.C., Hosking, J.G., Lutteroth, C. and Weber, G. Realistic Load Testing of Web Applications, In *Proceedings of the 10th European Conference on Software Maintenance and Re-engineering*, Berlin, 22-24 March 2006, pp 57-70.

DOI: [10.1109/CSMR.2006.43](https://doi.org/10.1109/CSMR.2006.43)

Abstract: We present a new approach for performing load testing of web applications by simulating realistic user behaviour with stochastic form-oriented analysis models. Realism in the simulation of user behaviour is necessary in order to achieve valid testing results. In contrast to many other user models, web site navigation and time delay are modelled stochastically. The models can be constructed from sample data and can take into account effects of session history on user behaviour and the existence of different categories of users. The approach is implemented in an existing architecture modelling and performance evaluation tool and is integrated with existing methods for forward and reverse engineering.

My contribution: Developed some of the key initial ideas for this research, co-designed approach, wrote the software the approach based on, wrote substantial parts of the paper, co-lead investigator for funding for this project from FRST

Realistic Load Testing of Web Applications

Dirk Draheim
Institute of Computer Science
Freie Universität Berlin
Takustr.9, 14195 Berlin, Germany
draheim@acm.org

John Grundy, John Hosking,
Christof Lutteroth, Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
{john-g.john,lutteroth,gerald}@cs.auckland.ac.nz

Abstract

We present a new approach for performing load testing of web applications by simulating realistic user behaviour with stochastic form-oriented analysis models. Realism in the simulation of user behaviour is necessary in order to achieve valid testing results. In contrast to many other user models, web site navigation and time delay are modelled stochastically. The models can be constructed from sample data and can take into account effects of session history on user behaviour and the existence of different categories of users. The approach is implemented in an existing architecture modelling and performance evaluation tool and is integrated with existing methods for forward and reverse engineering.

1 Introduction

Web applications are ubiquitous and need to deal with a large number of users. Due to their exposure to end users, especially customers, web applications have to be fast and reliable, as well as up-to-date. However, delays during the usage of the Internet are common and have been the focus of interest in different studies [2, 6]. The demands on a web site can change very rapidly due to different factors, such as visibility in search engines or on other web sites. Load testing is thus an important practice for making sure a web site meets those demands and for optimizing its different components [1]. Continual evolution of web applications is a challenge for the engineering of this class of software application. After each maintenance cycle, a convincing performance test must include a test of the full application under realistic loading conditions. In [16] it is recommended that load testing of a web site should be performed on a regular basis in order to make sure that IT infrastructure is provisioned adequately, particularly with regard to changing user behaviour and web site evolution. Also other experiences

stress the importance of load testing for the prediction and avoidance of service-affecting performance problems [23] at earlier stages of a web site's life cycle.

However, realism in the simulation of user behaviour for the purpose of load testing has been found to be crucial [29, 1]. "A load test is valid only if virtual users' behaviour has characteristics similar to those of actual users" because "failure to mimic real user behaviour can generate totally inconsistent results" [16]. Most current tools for load testing support the creation of simple test cases consisting of a fixed sequence of operations. However, in order to give the generated load some variety it is usually necessary to modify and parametrize these test cases manually. This is usually both time-consuming and difficult. A more elaborate approach is needed in order to generate a realistic load, and such an approach requires more advanced tool support.

Our approach applies the methodology of form-oriented analysis [9], in which user interaction with a submit/response style system is modelled as a bipartite state transition diagram. The model used in form-oriented analysis is technology-independent and suitable for the description of user behaviour. It describes what the user sees on the system output, and what he or she provides as input to the system. In order to simulate realistic users we extend the model with stochastic functions that describe navigation, time delays and user input. The resulting stochastic model of user behaviour can be configured in different ways, e.g., by analyzing real user data, and can be used to create virtual users of different complexity. The level of detail of the stochastic model can be adjusted continuously. All this enables our load test tool to generate large sets of representative test cases. Furthermore, our load test tool has its natural place in a chain of tools that support software engineers working on web applications. Those other tools also use the form-oriented model and thus help a software engineer to reuse or recover a web site's model for load testing.

Sect. 2 gives an overview of the methodology of form-oriented analysis that we use throughout the paper. Sect. 3

explains how this methodology can be applied in order to perform realistic simulation of web site users. Sect. 4 provides some information about how load tests are actually performed and delineates the whole picture of load testing; Sect. 5 explains the parameters used to describe workloads. Sect. 6 discusses related work, and Sect. 7 concludes the paper.

2 The Form-Oriented Model

Form-oriented analysis [9] is a methodology for the specification of ultra-thin client based systems. Form-oriented models describe a web application as a typed, bipartite state machine which consists of *pages*, *actions* and transitions between them. Pages can be understood as sets of *screens*, which are single instances of a particular page as they are seen by the user in the web browser. The screens of a page are conceptually similar, but their content may vary, e.g., in the different instances of the welcome page of a system, which may look different depending on the user. Each page contains an arbitrary number of *forms*, which in turn can have an arbitrary number of *fields*. The fields of forms usually allow users to enter information, and each form offers a way to submit the information that has been entered into its fields to the system. A submission invokes an action on the server side, which processes the submitted information and returns to the client a new screen in response. Hyperlinks are forms with no fields or only fields that are hidden to the user.

Form-oriented models can be visualized using *formcharts*. In a formchart the pages are represented as ovals and the actions as boxes, while the transitions between them are represented as arrows, forming a directed graph. Formcharts are bipartite directed graphs, meaning that on each path pages and actions occur alternately. This partitioning of states and transitions creates a convenient distinction between system side and user side: the page/server transitions always express user behaviour, while the server/page transitions always express system behaviour.

In Fig. 1 we see the formchart of a simple home banking system, which will be the running example of this paper. The system starts showing page Login to a user, who can enter an account number and access code. This data is submitted to action Verify, which checks if it is correct and either redirects the user back to the Login page or to the Menu page of the home banking system. Here the user can access the different functions, i.e., showing the account's status, making transfers, trading bonds, and logging out. Each of the functions may involve different subsystems and make use of different technical resources.

A form-oriented model of a web application offers several benefits. It is suitable for testing as well as for the analysis and development of dynamic web applications. A

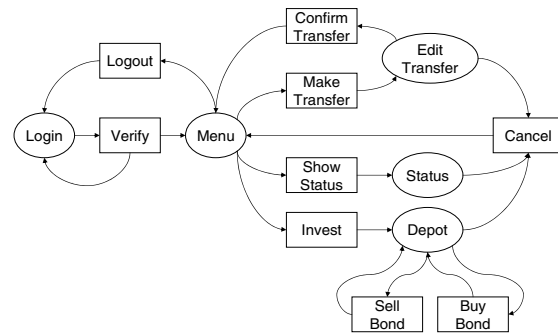


Figure 1. Formchart of example home banking web application.

typical shortcoming of many other models is that they do not capture fan-out of server actions, i.e., the ability of a server action to deliver many conceptually different client pages, which is covered by the form-oriented model.

3 Modelling User Behaviour with Stochastic Formcharts

Formcharts specify web applications, which usually work in a strictly deterministic manner. In a load testing scenario, however, the web application already exists, and the problem is to simulate the behaviour of a large number of users. But just as a formchart is a specification of the web application, it is also a specification of possible user behaviour; and while it is the web application that chooses in an action which page will come next and which data will be shown on the page, it is the user who chooses which of the available actions will be invoked afterwards and which data the action will get. In other words, when simulating users we have to model their navigational choices and the input they enter. Since we are aiming at real-time simulation, we also need to model the timing of user behaviour. In the case of web applications, this can be reduced to a model of user response time or “think time”, i.e., the time delay between reception of a screen and submission of a form. Since the fine-grained interaction involved in user input happens at the client side, transparent to the server, we do not model it.

We cannot predict user behaviour as we can predict the behaviour of a web application. Therefore, we use a stochastic model, which makes only assumptions about the probability of a particular user behaviour and not about which behaviour will actually occur. When estimating such probabilities, it can be important to take into account the session history of a user, which may influence the decision about the next step. For example, a user that has just

logged into the system is unlikely to log out immediately afterwards, but much more likely to log out after he or she did other things. Consequently, we are dealing with conditional probabilities.

The essential decision that a user makes on every page is about which form he or she will use. This affects much of the behaviour that will follow, so in our simulation we should make this decision first. There is always a limited number of forms on a page, and the question is how probable it is for each form to be chosen. This is expressed by probability distribution P_{form} , which also takes into account the session history. $Histories$ is the set of all possible session histories, and $Forms$ is the set of all forms. A form that is not available at a certain point in session history has probability 0; if there is only a single form available, it has probability 1.

$$P_{form}: Histories \times Forms \rightarrow [0, 1]$$

Once a form is chosen, we estimate a delay. This is done with p_{delay} . Time is a continuous variable, therefore p_{delay} is not a discrete probability distribution but a probability density function.

$$p_{delay}: Histories \times Forms \times (0, \infty) \rightarrow \mathbb{R}^+$$

Again, we acknowledge that the session history may have an influence on delay time, but we expect this effect to be much weaker than the effect of history on form choice. Therefore, we might simplify our model by neglecting session history. In our example, the delay probability density graphs for the forms on page Menu that lead to actions Logout, Status, Make Transfer and Invest, respectively, could look like the ones in Fig. 2.

One of the more complex tasks is the generation of input data for the fields of the chosen form. P_{input} is a discrete probability distribution that describes the probability that certain data is entered into the form. $Data$ is the universe of possible data, and if some particular data in it cannot be entered into a form, the value of P_{input} is 0.

$$P_{input}: Histories \times Forms \times Data \rightarrow [0, 1]$$

Also P_{input} depends on the session history. If, for example, the user needs a secret one-off transaction number (TAN) to add additional security to each transaction, such a TAN is only used once, hence the probability for that TAN to be used again shrinks. However, usually we can neglect the effect of session history on user input and use some simple logic instead in order to cope with such dependencies. Approaches for the generation of form input exist and have been discussed, for example, in [8, 3, 7].

Depending on the data we have about real user behaviour, there are different possibilities for us to configure the user model. In some cases, e.g., when the system we want

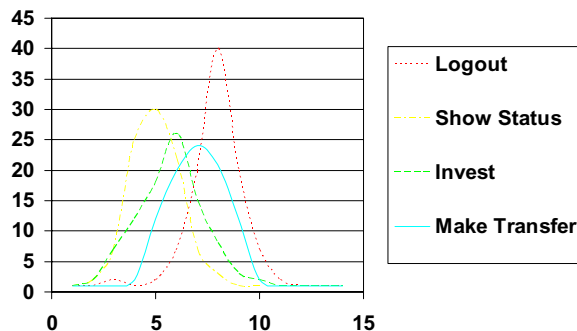


Figure 2. Probability densities for the delay caused with different forms.

to test is new, there might not be any sample of real user data yet. But when the system has already been used for some time there might be plenty of empirical historical user behaviour data. In both cases we try to make the user model as realistic as possible.

3.1 Repetition Models

After surveying real user session data, we can use it directly by replaying it on the system. Of course, the system would usually have to be reset to its original state first because it is not generally possible to play a session, e.g., for selling a bond, twice. We can adjust the load of the system by changing the time interval in which the sessions are replayed, or, when a session can be replayed an arbitrary number of times, the frequency of repetition. Form-oriented analysis offers suitable concepts for storing and representing session histories.

For surveying user behaviour we need a suitable instrument. In [8] we suggested the Revangie tool that can “snoop” on the communication between clients and server. Such a tool could be used for recording real user behaviour. Alternatively, server logs can be used. The problems and procedures of collecting real user data, and the benefit in the context of test case generation for functionality testing has been discussed, for example, in [14].

3.2 History-Free Stochastic Models

As discussed in the previous section, we can simply repeat real user sessions. But we can also exploit real user data in order to find suitable parameters for a stochastic model. A stochastic model is a more general approach and therefore more versatile. It is much easier to create new load testing scenarios by adjusting model parameters. Furthermore, replaying recorded data might not cause a system

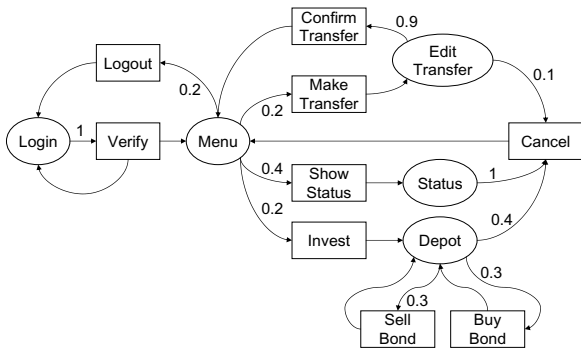


Figure 3. Simple stochastic formchart for the home banking system.

load that is representative. To create a representative repetition model requires a random sample of real user sessions with a sufficient size. If this is not given, the model might simply fail to cover the variety of possible input behaviour well enough, which might result in flaws of the system remaining undetected. In contrast to that, load testing with a stochastic model uses a randomized algorithm, which is generally less prone to yield tendentious results.

If we have data about how many times each form has been submitted, i.e., the total usage frequency of each form, we can use this to approximate P_{form} . The probability of a form to be chosen is set to the relative usage frequency, i.e., the number of usages of that form divided by the sum of the numbers of usages for all the forms on that page. This neglects the effect of session history on P_{form} and produces a stochastic formchart like the one in Fig. 3. In this formchart all page-action transitions are annotated with a transition probability, which reflects function P_{form} . At each page, the probabilities of all outgoing transitions sum up to 1 or possibly a bit less, with the remaining probability reserved for abrupt termination of the session. The transitions from actions to pages are performed by the system and therefore need no annotation. The problem of choosing transition probabilities for similar stochastic models is also discussed in [13, 27].

Such stochastic formcharts are similar to Markov chains, but there is a subtle and important difference: while a Markov chain creates a state machine with probabilities at every transition, a stochastic formchart is a bipartite state machine with probabilities only at the transitions going from page to action. Which transition will be chosen from action to a page is determined by the logic of the system, which is well-defined. Consequently, it makes no sense for load testing to model also this aspect stochastically and add probabilities to the action-page transitions, too. We cannot get rid of action-page transitions because an action can have

more than one outgoing transition, such as the action Verify.

3.3 History-Sensitive Stochastic Models

When we have samples of real user sessions and not just unrelated usage frequencies, we are able to create an empirical model that takes into account the effects of session history on P_{form} . A good method to define P_{form} with the help of this data is a decision tree (see, for example, [12]), which captures the relation between past events and future ones. Each path in the tree is a sequence of pages and actions, alternating, and represents a possible user session of a certain length. If there are cycles in the original formchart of a system, a corresponding decision tree can have arbitrary depth, and actions and pages of the original formchart can occur multiple times. In the decision tree we distinguish these multiple occurrences of actions and pages by giving them running indexes. All these actions and pages with index but the same name correspond to a single action or page in the original formchart.

Look, for example, at Fig. 4, which shows a possible decision tree for our home banking system. The root of the tree represents the state in which the system starts, i.e., page Login. Since it is the first occurrence of this page in our tree, we add the index 1 to its name. There is only one form on page Login, i.e., the form to enter account number and PIN. Logically, the probability that action Verify, which is invoked by that form, is chosen is 1 (or a little bit less if we would consider abrupt termination). The next two outgoing transitions of Verify are action-page transitions and therefore need no probability, as we have discussed in Sect. 3.2. The first of these transitions is chosen by the system when the authentication failed and leads back to page Login₁. This page, Login₁, is the same as the root of our tree. Formcharts allow us to visually represent actions and pages arbitrarily often, which can be good to avoid ugly transition arrows crossing over the diagram. If we want a page or action to have a certain transition, we can add a corresponding arrow to any of its corresponding bubbles/rectangles. The fact that there is a transition from Verify₁ back to Login₁ signifies that if the system chooses to go back to page Login, the user will, with regard to P_{form} , behave stochastically just the same as when he or she first entered the system at that page. This equivalence of user behaviour also includes future behaviour, i.e., the probabilities of form choices on pages to come. We need this recurrence to states in order to handle cycles in the original formchart, which could not be represented with a finite decision tree otherwise. On the arrows representing the outgoing transitions of page Menu₁ we see the probabilities with which the user chooses different forms just after he or she logged in. Let us have a closer look at what happens when the user chooses the form that leads to ac-

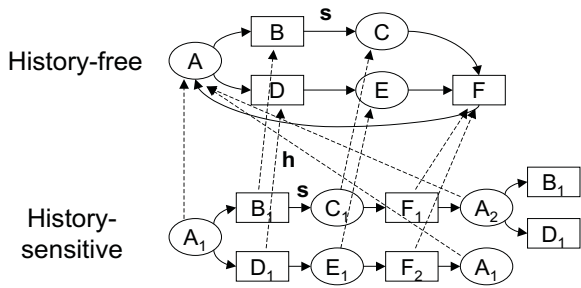


Figure 5. Homomorphism from history-sensitive to history-free formchart model.

tion Invest. Page Depot₁ offers a form for action Cancel₁, which means that a simulated user that invokes it will come back to page Menu₁ and stochastically behave like a user who has just logged in. There are also forms for actions Buy Bond and Sell Bond, which are assigned equal probabilities. When action Buy Bond is invoked, we get back to page Depot. However, the probabilities have changed, and we model this situation with a new page Depot₂. The probability for the invocation of Cancel is now higher, i.e., once a bond has been bought, the user is more likely to go back to Menu. Furthermore, the probability to buy another bond is now higher than the probability to sell one, which might reflect, for example, the common pattern that new bonds are bought from the money old ones were sold for, but not vice versa. In all the three possible cases we refer back to other parts of the decision tree, which means that the following pattern of user behaviour is already part of the model. However, if we found that after some action the user behaviour differs significantly from what has already been modelled, we would add another layer of pages and actions to the tree that reflects these differences. Following this pattern, the effects of session history in P_{form} can be approximated arbitrarily.

It is important to note that the relation between a system's original history-free formchart and a corresponding history-sensitive stochastic formchart model is formally well-defined. There exists a *homomorphism* h between the pages and actions U of a formchart that takes into account session history and the pages and actions V of the corresponding one that does not. $h:U \rightarrow V$ maps all pages or actions $X_i \in U$ to the corresponding page or action $X \in V$. Any sequence of action invocations s performed from a certain page $u \in U$ that ends at a page $s(u) \in U$ can also be performed on page $h(u) \in V$ and will end at page $h(s(u)) \in V$. This relation is illustrated in Fig. 5.

3.4 Multimodels

Within the population of all users there are usually different categories of users that use some features of the web application particularly often, which causes differences in P_{form} , or with particular skill, which generally reflects in p_{delay} . The presence of distinct groups of novice users and expert users, for example, can be visible in the fact that the graph of p_{delay} has more than one peak.

In order to find out which categories exist, we can analyse data about real user sessions and perform a classification. One method for doing this automatically is clustering (see, for example, [12]). When we have divided the sessions into different categories, we can create a user model for each category using the respective subset of data. Once we have a set $Models$ of different user Models, we can define a distribution P_{model} of the probability that a user is of a certain category and has thus to be simulated with the corresponding model.

$$P_{model}: Models \rightarrow [0, 1]$$

This distribution can simply be inferred from the relative sizes of the different categories. In order to produce a realistic load during load-testing, we stochastically choose a user model every time we create a new virtual user. When load-testing models incorporate several user models, we call them *multimodels*.

Such multimodels enable prioritization techniques proposed for functional testing, e.g., as discussed in [10]. Such techniques, which execute more important test cases more often than others, can also be reasonable from a load testing perspective, for example for making sure that particularly popular or critical parts of a web site are tested thoroughly. Such a multimodel can also be understood as a complex form of operational profile [18] that takes into account different types of user behaviours with their respective occurrence probabilities. In certain circumstances, multimodels can be subsumed under history sensitive formchart models.

4 Performing Load Tests

In order to actually perform a load-test, we have to use one of the described user models to create a *workload* on the system, i.e., an activity that consumes some of its resources. For this task, we can choose between different workload models, which describe workload over time. In order to push a system to its limits we could, for example, utilize the increasing workload model, which adds more and more virtual users to the system. The parameters that can be used to describe a workload are explained in Sect. 5. It is important that the load test environment the system is running in is very close to the production environment, i.e.,

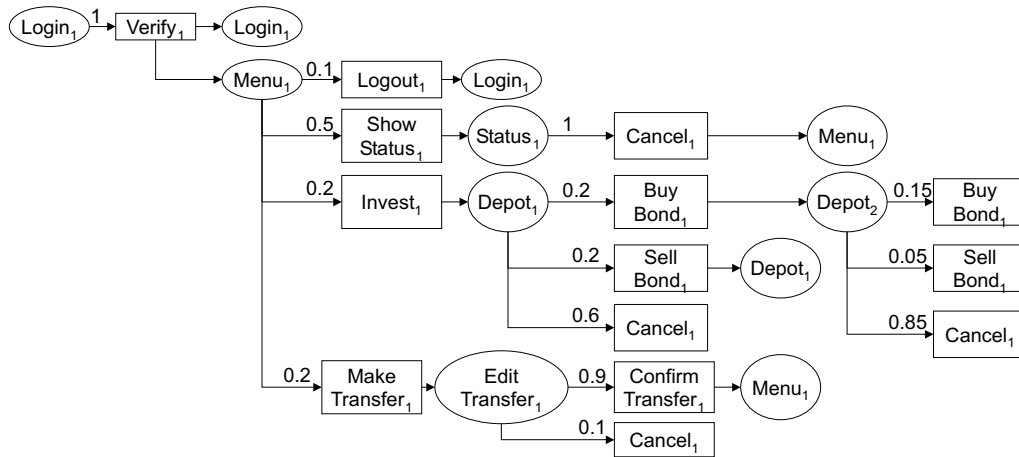


Figure 4. History-sensitive formchart model for the home banking system based on a decision tree.

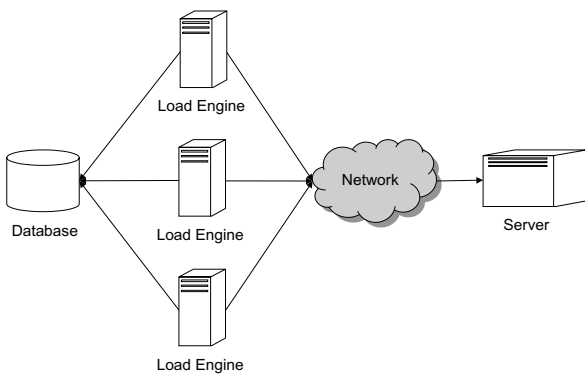


Figure 6. Load-testing environment setup.

the one the system is eventually intended to work in. The most accurate results will be produced when the load test is performed in the production environment itself, although this is usually not possible in a system that is being used already. As one can see in the schematic illustration of the load testing environment in Fig. 6, we perform the load-test with the help of dedicated computers, which are called *load engines*. The load engines simulate virtual users and write data about the state of the testing environment and its measured performance to a database. Another important requirement of the test environment is that the load engines themselves are bottleneck-free. If this is not the case, then performance measurements of the server will be distorted by delays introduced by the load engines [1]. That is why, from a certain number of virtual users on, we distribute the simulation of the users onto several load engines.

Now let us consider what each of the virtual users actually does. According to the model, an initial page request is sent to the system. The load engine lets the virtual user wait

for a random time given by p_{delay} . Then, a form and corresponding input is chosen randomly by P_{form} and P_{input} , respectively. This information is used in order to create and send a new request. Since the server action corresponding to the chosen form may generate screens of different pages, the received screen has to be classified in order to determine which of the possibly many action-page transitions was chosen by the system. How such classification can be done has been described, for example, in [8]. Once the classification is done, the load engine can again simulate a delay. Another aspect of virtual users worth mentioning is that it can sometimes be necessary to use a managed set of virtual user profiles. This means maintaining and using data that cannot be generated randomly, e.g., account numbers and corresponding PINs for our home banking example. These data can also be stored in the database all the load engines are connected to.

4.1 Finding Performance Bottlenecks

When looking for system bottlenecks, we usually observe the system in a state of extreme load, which is also called a *stress test*. An important measure to keep track of is the roundtrip-response time [16], i.e., the time between the request of a load engine to the server and the reception of its response, because this reflects the time a real user would have to wait. Also the response time on the server, i.e., the time between the reception of a request and the sending of a response, is usually monitored. Other common measures are, for example, the CPU and memory usage on the server. More about metrics for load testing can be found in [21].

A common way to perform stress testing is to use an increasing workload model and add virtual users until the response time crosses a threshold that we consider long enough to render the system unusable, i.e., longer than a

user would probably be willing to wait. With the data collected by the load engines we are able to analyse the response time of individual actions. The number of users where the response time crosses a certain limit usually differs between the actions, depending on how much they are invoked and the technical resources they are driven by. Actions for which this limit is crossed early present a bottleneck of the system and usually allow us to draw conclusions about the underlying subsystem.

4.2 Load Testing of Legacy Systems

Often we face the task of load testing a legacy system, i.e., a system which is already deployed and running, and for which the information necessary to create a realistic user model is not available. In such a case we could either use a very simple user model, such as a generic one that invokes actions randomly with a uniform distribution, or try to extract the necessary information by means of reverse engineering. In [8] we proposed a methodology and a tool called Revangie which is able to reconstruct form-oriented analysis models for existing web applications. Models can be constructed online, i.e., during system exploration, but also offline, e.g., from recorded user data. There also exist other tools for model recovery of web sites that can be useful for the creation of load models, e.g., the ones described in [3].

4.3 Load Testing in the MaramaMTE Tool

Our implementation extends earlier work on performance estimation of distributed systems generated using the ArgoMTE tool. This tool generates testbeds for such systems from descriptions of their software architecture allowing performance estimation to be carried out at design time and lessens the cost of experimenting with multiple software architecture choices [5]. This work has recently been ported to an Eclipse-based implementation in the form of the MaramaMTE tool. A screenshot of MaramaMTE in use specifying a software architecture is shown in Fig. 7. In this example, the architecture modelled consists of a set of RMI remote objects (CustomerManager, UserManager, AccountManager) and database tables (customer, user, account). MaramaMTE permits specification of client-based testing from the architecture description using the simple assumption that internal code within modules is much lower cost than inter-module communication (valid for most applications; note if actual code has been implemented for a module this can be used in place of the testbed generated code). Clients can be multi-threaded and client load can be repeated; in this example ClientTest1, ClientTest2 are repeated each 1000 times. While this tool is very successful, it lacks the ability to model user interaction appropri-

ately. Accordingly, we have implemented our load testing approach for the history free stochastic model in the MaramaMTE toolsuite.

Our implementation adds a formchart view to MaramaMTE for editing the model (Fig. 8). The test designer can define the interaction model and annotate the transitions with probabilities. Delay distributions can be defined as described earlier. For the load testing process the test designer can specify a workload model. The actions specified in the formchart view are linked to remote service calls specified in the MaramaMTE architecture views, combining the two models together; for example the login_Test2 page in Fig. 8 is linked to the findUser service of the UserManager component in Fig. 7. The load test can be activated from the MaramaMTE toolsuite. This toolsuite possesses a sophisticated remote deployment functionality; the tool can automatically deploy load test agents acting as clients on different machines and orchestrate their activity. From the MaramaMTE tool, the whole load test architecture can be controlled.

Combining our stochastic model-based approach with MaramaMTE provides a very powerful model-based performance estimation approach where realistic estimates of a web application's performance can be performed at design time before significant implementation expenses have occurred.

5 Workload Models

A workload is completely described by a user model and one of several possible workload parameters; both elements together give a workload model. A user model is a stochastic formchart with user delays. A workload parameter can be a function over time, modelling a changing workload. For the introduction and comparison of the different parameters here we focus on constant parameters. We will introduce workload parameters that are partly based on virtual users, partly on the concept of user sessions. A *user session* is the model of what we consider as one typical connected usage of the system by a user; we assume that users always explicitly start and end their sessions. User sessions can be modelled by identifying the session delimiting transitions like login and logout in a formchart; the user session model is then a part of the user model. Given a stochastic formchart, a session model may have a defined *average number of requests (AVGR)* before the session terminates; note that not every distribution has to have such an expected value. But it is possible to statically check whether a stochastic user session model has such a finite *AVGR*; then we call it a finite user session. The *client request rate (CRR)* is the request rate of the individual virtual user. It is determined by the server response time and the user think time after receiving the response. If this think time is kept con-

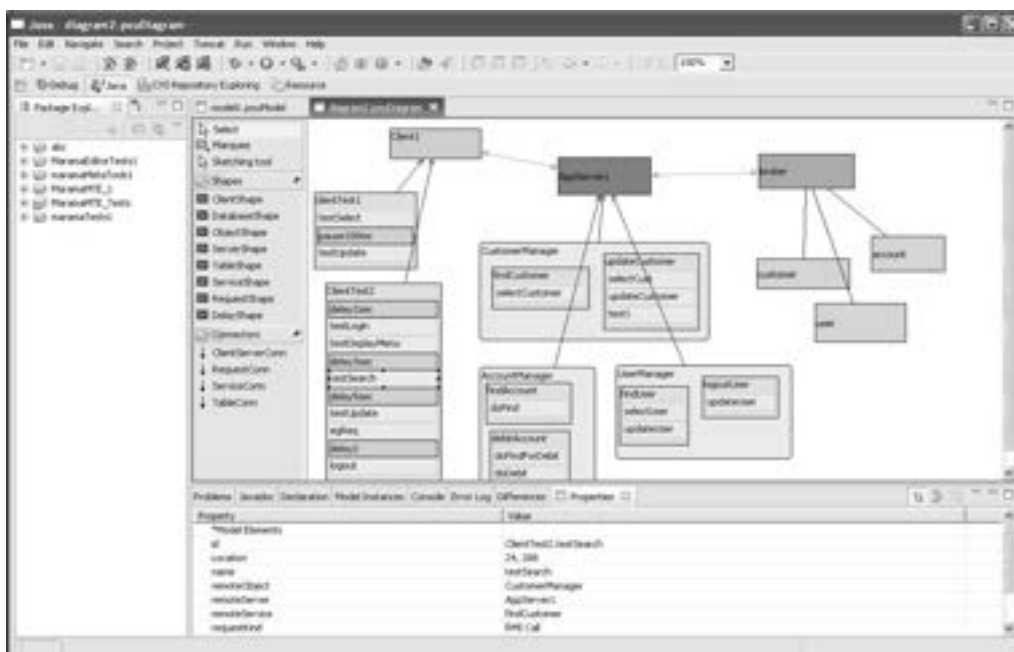


Figure 7. Screenshot of the architecture view of the MaramaMTE tool.

stant, then CRR decreases as soon as the server response degrades. For load testing tools, a load-independent CRR is often recommended, but this requires in general a non-trivial implementation [1]. We will also consider the *average session duration (AVGD)*. This value takes into account server response times as well as think times. If the server response times are negligible, then exactly the finite user sessions have a defined $AVGD$.

There are two classical workload parameters, one is the *number of virtual users (VU)*, that is the number of user processes active at a point in time. Another workload parameter is the *request rate (RR)*, that is the number of requests generated per time unit. All client requests — in our terminology form submissions — are counted. For workload models with finite user sessions we introduce a different workload parameter, the *starting user session rate (SUR)*. This is the average number of finite user sessions that is started per time unit. All three parameters can be used to describe a constant load. If VU is the workload parameter, a constant load is achieved by generating a certain number of virtual users, and then ceasing to generate new users. We can also control the load with SUR . A constant load is achieved by continuously generating new user sessions with a constant SUR . After an initial start-up time in the order of $AVGD$ we have constant load. VU is in this case an observable parameter that is affected by SUR and other parameters.

The different elements of the workload model, user

model and workload parameter should model truly different aspects of the model as a separation of concerns; otherwise the model will be unrealistic as we will see in the following. We define: a workload description is *realistic* if the following holds for a change in the user model. If an element of the user model is changed, for example if a think time is shortened, perhaps in order to model an improved page readability, or if the number of requests per user session is changed, perhaps because the user navigation is improved, then this change in the user model should have the same effect on the load test as it would have on the real system. We restrict our definition here to these two types of changes in the user model. We can show that the two major conventional workload parameters, namely VU and RR , do not give realistic workload models, but SUR does.

We now discuss whether a workload model with workload parameter VU is realistic. If the server response times are negligible, we note that for such models globally scaling all think times by factor a ceteris paribus changes RR by factor $1/a$, because each virtual user delivers a changed CRR ; obviously we have

$$RR = VU \cdot CRR$$

The scaling of think times hence changes the system load in the load test. On the real system however the load is not expected to change as we will see soon; if all users simply take a little bit longer to think, but still do the same number of requests, RR does not change! Hence realism is vio-

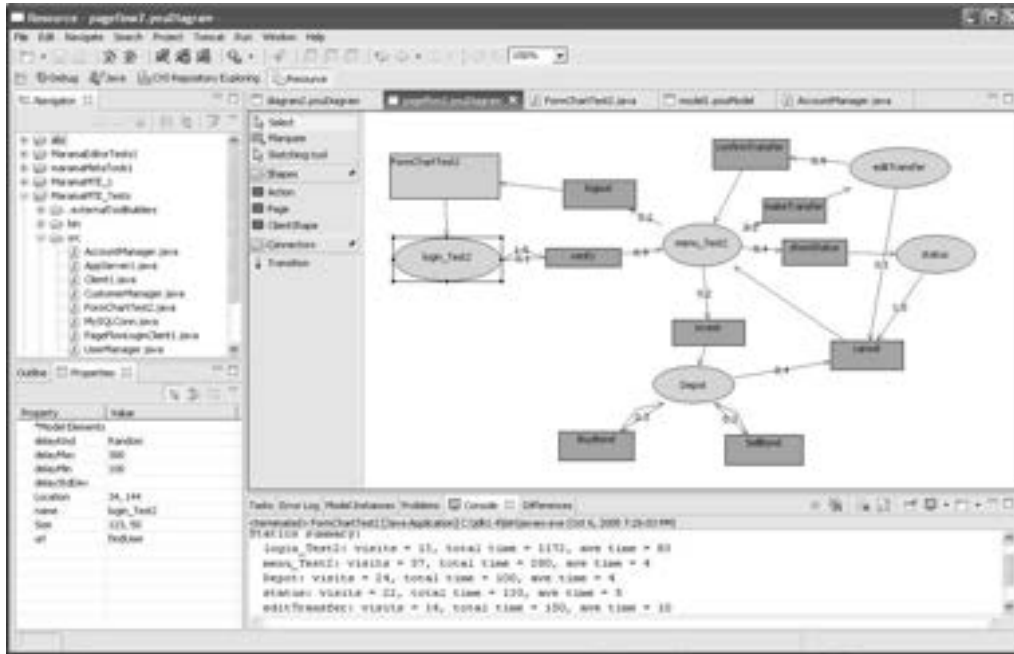


Figure 8. Screenshot of the formchart view of the MaramaMTE tool.

lated for a workload model with workload parameter VU . We now discuss whether a workload model with workload parameter RR is realistic. We note that for such models the request rate is trivially kept constant if think times are scaled. We now consider the second condition of realism; if we change the user model by reducing $AVGR$ and we assume for simplicity that all requests create the same load, then the load during the load test remains constant if we have fixed RR . But on the real system, the load would decrease. Hence workload models with workload parameter RR are unrealistic.

We now consider a workload model with workload parameter SUR . After an initial start-up we have, as one easily convinces oneself:

$$RR = SUR \cdot AVGR$$

Hence globally scaling all think times *ceteris paribus* does not change RR , since the duration of the single user session is irrelevant after start-up: The equation contains $AVGR$ and not $AVGD$. This behaviour of the load test is exactly the behaviour of the real system; for the real system, the same equation holds. We now consider the second condition of realism; if we change the user model by, say, halving $AVGR$, and if we assume for simplicity that all requests create the same load, then in both, the load test and the real system, the load will be halved. According to our definition this indicates that workload models with workload parameter SUR are realistic. In fact, workload models with

workload parameter SUR have other advantages. Furthermore RR is not sensitive to server load, even if the think time of the session clients would be sensitive to the server load. Even if the actual user agents are programmed in a way that they have $AVGD$ that are dependent on the server response, if SUR is kept constant, then the load is constant. This is because VU is changed appropriately if we scale the think times; in fact it is easy to see that we have:

$$VU = SUR \cdot AVGD$$

We discuss now an example showing that an unrealistic workload model, if applied naively, can create misleading load test results. We take an example from one of the load test projects the authors are involved with. It is an enrolment system for university students. Using the system is mandatory for students. We compare now the behaviour of the different load test approaches during maintenance of the application. We assume a workload model with workload parameter RR . We assume the system performance is sufficient. We assume every student performs 10 requests for his enrolment, only the last one creates heavy load throughout the system. Now the user interface is improved, and only two requests per user are necessary, a first lightweight one and the last one being the same heavy load request as before. If we naively change only the user model in the workload model, and not the workload parameter, then the system load would increase roughly by factor 5 and could bring down the system. In the real system the system load

would not increase. The problem remains if the workload model is set up with workload parameter VU . In contrast, if the workload model is set up with the realistic workload parameter SUR , the system load remains roughly the same in the load test, hence resembling the behaviour of the running application. As we see, workload models with workload parameters RR or VU can deliver spurious results, if the parameters are not changed with every change to the user model. These changes can be done, but they require the knowledge of the above equations or lucky intuition to the same effect. The correct load test behaviour comes for free in the workload model with the realistic workload parameter SUR .

6 Related Work

In many cases, load-testing is still done by hand-written scripts that describe the user model as a subprogram [22, 25]. For each virtual user, the subprogram is called, possibly with a set of parameters that describe certain aspects of the virtual user's behaviour. Often the users are also modelled by a multimodel, which defines a subprogram for each user category. Data is either taken from a set of predefined values or generated randomly. With regard to input data this approach has a certain degree of randomization. However, user behaviour itself mainly remains a matter of repetition. This approach is purely script driven and suffers, like any hand-written program, from being prone to programming errors. The load engine itself has to be developed and brought to a mature state, which usually is a very time consuming task. In contrast to that, our approach does not rely on hand-written programs but on configurable models, which are much easier to handle and less error prone. It does not require a rewrite of the load engine itself, but rather a reconfiguration of a load engine that interprets stochastic formcharts.

The leading product for industry-strength load testing [19] is Mercury Interactive's LoadRunner [17]. It takes a similar script-driven approach. However, it significantly increases usability by offering a visual editor for end-user scripts. No conventional programming is needed, and the scripts describe the load tests in a much more domain-specific manner. End-user scripts are run on a load engine that takes care of load balancing and monitoring automatically. LoadRunner does not, however, offer a model-based solution like that of stochastic formcharts. In contrast to the LoadRunner tool suite, which focuses on load testing and optimisation, formchart models offer well-understood concepts for the specification of systems in general, which are also useful for other web application engineering tools and facilitate their interoperability. Most current load testing tools operate in a manner similar to LoadRunner. A detailed discussion of bottleneck problems created among

other things by operating systems is given in [1]. Additionally the authors present a nontrivial implementation for load test clients. More information about load test practice can be found in [24, 11, 16].

There already exist model-based approaches for testing of web applications, e.g., in [15, 26], but they usually focus on the generation of test cases for functionality testing. Different studies have shown that stochastic models, in particular Markov chains, provide benefits for functionality testing [26, 13, 27]. They can be used for the automatic generation of large randomized test suites with a high coverage of operational paths. In [20, 26], for example, analysis models are used for regression testing in web site evolution scenarios. The model for user navigation is a stochastic one similar to Markov chains, but all user input data has to be given in advance for the system to work. While this may be appropriate for regression testing, it is not flexible enough for performing load tests. A Markov chain model like that in [27, 26] can only be used for a system where identical inputs cause identical state transitions, which is not the case in most web applications that rely on session data or a modifiable database. Consider, for example, an online ticket reservation system: after a specific place has been booked, it is not available any more; thus, repeating the same inputs will cause different results.

The motivation of using a statistical model based on data about real user behaviour for realistic load testing of web sites was already anticipated in [13], but their model fails to distinguish the user behaviour, which can only be adequately modelled as a stochastic process, from the system behaviour, which is deterministically given by the implementation. Transforming a state model of a web site directly into a Markov chain is not sufficient and does not account for the system's behaviour, which is not stochastic. In [28] it was shown that creation of a simple stochastic user model with real user data represents a valid approach for load testing. However, most approaches offer no model for specifying user behaviour over time, and it is usually neglected that form choice probabilities may change during a session.

The difference to stochastic process calculi such as, for example, the one described in [4], is that our stochastic model captures the specifics of web applications and can also be very suitably used for web application development [9]. However, it is conceivable that such theory can help in the development of new analysis techniques for our stochastic models. Since action-page transitions are chosen by the system, it is not necessary to assign probabilities to them in order to perform load testing. But if we assign probabilities to these transitions the same way we do with page-action transitions, e.g., by measuring relative frequencies, estimation or simply using uniform probabilities, then we can analyse the model statically and make statistical estimates similar to those described in [27] for Markov chains.

The difference to Markov chains is that our model also captures behaviour over time, i.e., delays. So if we extend our model further by measuring or estimating time delay distributions for the server actions, then we could also make statistical estimates of timing behaviour. This could, for example, allow the calculation of the expected duration of a session or of the expected usage of particular subsystems. Such new analysis techniques present possible future work in the area of web site performance evaluation.

7 Conclusion

In this paper we presented a new approach for load testing of web sites which is based on stochastic models of user behaviour. It allows the easy creation of realistic models of the individual user behavior. Furthermore we discussed how the user model can be used in a realistic workload model. We described our implementation of realistic load testing in a visual modelling and performance testbed generation tool, which allows realistic estimates of web application performance from software architecture descriptions.

References

- [1] G. Banga and P. Druschel. Measuring the Capacity of a Web Server under Realistic Loads. *World Wide Web*, 2(1-2):69–83, 1999.
- [2] P. Barford and M. Crovella. Measuring Web Performance in the Wide Area. *SIGMETRICS Perform. Eval. Rev.*, 27(2):37–48, 1999.
- [3] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings of 11th International WWW Conference*, May 2002.
- [4] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: a Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theor. Comput. Sci.*, 202(1-2):1–54, 1998.
- [5] Y. Cai, J. C. Grundy, and J. G. Hosking. Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. In *Proceedings of the 2004 IEEE International Conference on Automated Software Engineering*, pages 36–45. IEEE Press, September 2004.
- [6] K. Curran and C. Duffy. Understanding and Reducing Web Delays. *Int. J. Netw. Manag.*, 15(2):89–102, 2005.
- [7] Y. Deng, P. Frankl, and J. Wang. Testing Web Database Applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.
- [8] D. Draheim, C. Lutteroth, and G. Weber. A Source Code Independent Reverse Engineering Tool for Dynamic Web Sites. In *9th European Conference on Software Maintenance and Reengineering*. IEEE Press, 2005.
- [9] D. Draheim and G. Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.
- [10] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.
- [11] A. K. Iyengar, M. S. Squillante, and L. Zhang. Analysis and Characterization of Large-scale Web Server Access Patterns and Performance. *World Wide Web*, 2(1-2):85–100, 1999.
- [12] K. Jajuga, A. Sokoowski, and H. H. Bock. *Classification, Clustering and Data Analysis*. Springer, August 2002.
- [13] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Trans. Softw. Eng.*, 27(11):1023–1036, 2001.
- [14] S. Karre. Leveraging User-Session Data to Support Web Application Testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005. Member-Sebastian Elbaum and Member-Gregg Rothermel and Member-Marc Fisher II.
- [15] D. C. Kung, C.-H. Liu, and P. Hsia. An Object-Oriented Web Test Model for Testing Web Applications. In *COMP-SAC '00: 24th International Computer Software and Applications Conference*, pages 537–542, Washington, DC, USA, 2000. IEEE Computer Society.
- [16] D. A. Menascé. Load Testing of Web Sites. *IEEE Internet Computing*, 6(4):70–74, July 2002.
- [17] Mercury Interactive Corporation. Load Testing to Predict Web Performance. Technical Report WP-1079-0604, Mercury Interactive Corporation, 2004.
- [18] J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, 1998.
- [19] Newport Group Inc. Annual Load Test Market Summary and Analysis, 2001.
- [20] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.
- [21] L. Rosenberg and L. Hyatt. Developing a Successful Metrics Program. In *Proceedings of the 19th International Conference on Software Engineering*. ACM Press, 1997.
- [22] A. Rudolf and R. Pirker. E-Business Testing: User Perceptions and Performance Issues. In *Proceedings of the First Asia-Pacific Conference on Quality Software*. IEEE Press, 2000.
- [23] J. Shaw. Web Application Performance Testing – a Case Study of an On-line Learning Application. *BT Technology Journal*, 18(2):79–86, 2000.
- [24] J. C. C. Shaw, C. G. Baisden, and W. M. Pryke. Performance Testing – A Case Study of a Combined Web/Telephony System. *BT Technology Journal*, 20(3):76–86, 2002.
- [25] B. Subraya and S. Subrahmanya. Object Driven Performance Testing of Web Applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software*. IEEE Press, 2000.
- [26] P. Tonella and F. Ricca. Statistical Testing of Web Applications. *Software Maintenance and Evolution*, 16(1-2):103–127, April 2004.
- [27] J. A. Whittaker and M. G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.
- [28] L. Xu and B. Xu. Applying Users' Actions Obtaining Methods into Web Performance Testing. *Journal of Software (in Chinese)*, (14):115–120, 2003.
- [29] L. Xu, B. Xu, and J. Jiang. Testing Web Applications Focusing on their Specialties. *SIGSOFT Softw. Eng. Notes*, 30(1):10, 2005.

4.5 A Domain-Specific Visual Modeling Language for Testing Environment Emulation

Liu, J., Grundy, J.C., Avazpour, I., Abdelrazek, M. A Domain-Specific Visual Modeling Language for Testing Environment Emulation, 2016 *IEEE Symposium on Visual Languages and Human-Centric Computing*, Cambridge, UK, Sept 4-8 2016, IEEE Press, pp. 143 - 151.

DOI: [10.1109/VLHCC.2016.7739677](https://doi.org/10.1109/VLHCC.2016.7739677)

Abstract: Software integration testing plays an increasingly important role as the software industry has experienced a major change from isolated applications to highly distributed computing environments. Conducting integration testing is a challenging task because it is often very difficult to replicate a real enterprise environment. Emulating testing environment is one of the key solutions to this problem. However, existing specification-based emulation techniques require manual coding of their message processing engines, therefore incurring high development cost. In this paper, we present a suite of domain-specific visual modeling languages to describe emulated testing environments at a high abstraction level. Our solution allows domain experts to model a testing environment from abstract interface layers. These layer models are then transformed to runtime environment for application testing. Our user study shows that our visual languages are easy to use, yet with sufficient expressive power to model complex testing applications.

My contribution: Developed initial idea for the research, co-developed tool design, co-supervised PhD student, wrote substantial parts of paper, investigator on funding of the project from the Australian Research Council

A Domain-Specific Visual Modeling Language for Testing Environment Emulation

Jian Liu
School of Software and Electrical Engineering
Swinburne University of Technology
Hawthorn, VIC 3122, Australia
jianliu@swin.edu.au

John Grundy, Iman Avazpour, Mohamed Abdelrazek
School of Information Technology
Deakin University, Burwood, VIC 3125, Australia
j.grundy@deakin.edu.au
iman.avazpour@deakin.edu.au
mohamed.abdelrazek@deakin.edu.au

Abstract—Software integration testing plays an increasingly important role as the software industry has experienced a major change from isolated applications to highly distributed computing environments. Conducting integration testing is a challenging task because it is often very difficult to replicate a real enterprise environment. Emulating testing environment is one of the key solutions to this problem. However, existing specification-based emulation techniques require manual coding of their message processing engines, therefore incurring high development cost. In this paper, we present a suite of domain-specific visual modeling languages to describe emulated testing environments at a high abstraction level. Our solution allows domain experts to model a testing environment from abstract interface layers. These layer models are then transformed to runtime environment for application testing. Our user study shows that our visual languages are easy to use, yet with sufficient expressive power to model complex testing applications.

Keywords—*model-driven engineering; domain-specific visual modeling language; software component interface description; testing environment emulation.*

I. INTRODUCTION

Enterprise software systems have become more complicated and interconnected to provide composite services to their consumers. The behavior of these systems are no longer governed by their own system components, but also driven by its increasingly complex interactions with other systems in its operational environment. This means that testing these interconnections in a realistic production environment is critical.

Many approaches have been proposed to provide executable, interactive representations of deployment environments, e.g. method stubs, mock objects and existing emulation solutions [1-5]. These approaches, however, introduce a large implementation overhead for developers, especially for large-scale heterogeneous environments. Consequently, we have developed a novel domain-specific Visual Modeling Language for Testing environment emulation (TeeVML) and tool support to reduce the development cost of testing environment. In this paper, we present a suite of three visual languages used for endpoint signature, protocol and behavior layers modelling.

978-1-5090-0252-8/16/\$31.00 ©2016 IEEE

The remainder of this paper is organised as follows: Section II motivates our work with an example case study. It is followed by an introduction of the approach and design of TeeVML in Section III. In Section IV, we show how a testing endpoint can be modeled and then describe the steps to convert endpoint models into testing runtime environment in Section V. In Section VI, we evaluate the tool and discuss the key findings from the results of a user survey. This is followed by a review of related work in section VII. Finally, we conclude this paper and identify some problems for future work in Section VIII.

II. MOTIVATION

We use an example case study of a bank system. Consider a new Internet banking system is to be integrated to a corporate banking environment. The bank has a core system to support in-house daily banking operations, servicing its customers. For the purpose of expanding its customer base and reducing operational costs, the bank plans to introduce an Internet banking system for allowing its customers to do bank account queries and transactions by themselves. Due to operational issues and data security considerations, all bank accounts and customer data will be kept in the core system. The Internet banking system must interact with the core system intensively for data exchange. To ensure interconnectivity and mutual operability between the core system and the new Internet banking system, integration testing must be conducted before putting the new system in place.

For this study, we treat the Internet banking system as System Under Test (SUT), and the core banking system as testing application (or call endpoint) to be emulated. The endpoint must provide interconnectivity and interoperability testing functionalities, which should mimic its real application services. Let's assume the main endpoint characteristics, as described below:

- An endpoint only considers the external behaviors (or call services) of the real system, and all internal implementations will be ignored;
- An endpoint only provides a subset of the real system invoked by the SUT;
- An endpoint should be able to detect all SUT interface defects, identify their types and origins.

Currently, Kaluta system creates testing endpoints by manually coding the endpoint using Java and Haskell languages [5]. A typical testing endpoint includes an endpoint type dependent message processing engine module, and an endpoint type independent network interface module. Thus, cost of emulating a testing environment will be linearly increased along with additions to endpoint types. UML Testing Profile (UTP) is a model-driven testing approach, providing a specification-based means to systemically define tests for static and dynamic aspects of systems modeled in UML [6]. However, the UTP is for server-side system testing, rather than developing a testing environment for client-side application integration testing.

III. OUR APPROACH

Our TeeVML is based on a new layered software interface description framework, and a suite of Domain-Specific Visual Languages (DSVL) are developed for modeling each interface layer of an endpoint. From a high-level point of view, our approach consists of an endpoint modeling environment supported by TeeVML and an Axis2 Web Service runtime environment. Domain experts work on the modeling environment to create endpoint models; and these models are transformed to target source codes automatically by code generators. Fig. 1 depicts TeeVML modeling and runtime testing environments.

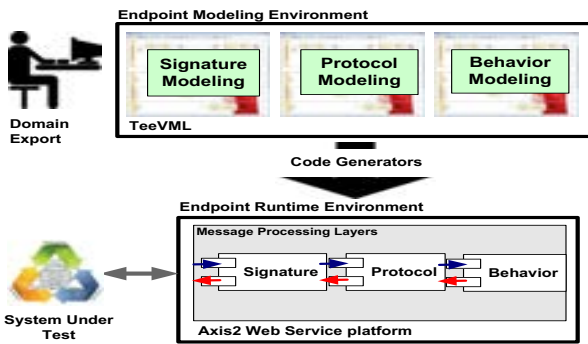


Fig. 1. TeeVML endpoint modeling and runtime testing environments

A. Software Interface Description Framework

To design our framework, we analyzed three popular applications to gather as much domain knowledge as possible, by identifying similar objects and operations. These applications were:

- A public cloud Customer Relationship Management (CRM) application for providing sales process automation; the CRM needs to be integrated with an in-house Enterprise Resource Planning (ERP) system;
- A LDAP server for enterprise resources management; normally, a new application must be integrated with an enterprise LDAP server first, before it can be put to use in production environment;
- An e-commerce sample application jPetStore, originally developed by Sun Microsystems for illustrating the usage of J2EE technology and best practices in system design.

From this domain analysis, we proposed a new layered software interface description framework, which is a

modification of Han’s comprehensive interface definition framework for software components [7]. Our framework abstracts software interface into three horizontal and two vertical layers. Horizontal layers include signature (message format and structure), protocol (valid service temporal sequence) and behavior (service request process and response generation). Vertical layers include data store (data persistence and manipulation) and Quality-of-Service (QoS) (or call non-functional requirement). A SUT service request is processed horizontally by the endpoint in step by step from signature, protocol, down to interactive behavior layer. Whenever an error occurs at any layer, the request process will be terminated.

The signature and protocol layers act as message pre-processors for checking service request syntax and sequence correctness, before handing it over to the behavior layer for generating response. Vertical layers are not directly involved in request processing, but provide support to horizontal layers. We use modular development approach to model an endpoint – i.e. each module represents a particular interface layer.

TABLE I. PON PRINCIPLE AND VISUAL NOTATION DESIGN RULE

PoN Principle	Description	TeeVML Visual Notation Design Rule
Semiotic clarity	There should be 1:1 correspondence between semantic constructs and graphical symbols.	All the visual symbols have 1:1 correspondence to their referent concepts.
Perceptual discriminability	Different symbols should be clearly distinguishable from each other.	All symbols use different variables, plus redundant coding by colours or textures.
Semantic transparency	Visual representations whose appearance suggests their meaning.	We have used as many icons as possible to represent visual symbols, and minimised the use of abstract geometrical shapes.
Complexity management	There should be some explicit mechanisms for dealing with complexity	Hierarchical visual presentations and information hiding are used to manage diagrammatic complexity.
Cognitive integration	There should be some mechanisms to support integration of information from different diagrams.	Service nodes in behavior model import request and response parameters from signature model.
Visual expressiveness	Use the full range and capacities of visual variables.	We have used various visual variables, such as shape, colour, orientation, texture, etc. when designing visual symbols.
Dual coding	Use text to complement graphics.	Most visual symbols have a textual annotation.
Graphic economy	The number of different graphical symbols should be cognitively manageable	A key design consideration is to minimise the number of visual symbols.
Cognitive fit	Use different visual dialects for different tasks and audiences	Not applicable.

Visual notations form an integral part of a domain-specific visual language, and have a profound effect on the usability and effectiveness of the visual language [8]. To evaluate the “goodness” of visual notations, Larkin et al. defined the cognitive effectiveness [9] as “the speed, ease, and accuracy with which a representation can be processed by the human mind”. To achieve the cognitive effectiveness, Moody proposed

the Physics of Notations (PoN) [8], and defined a set of principles to evaluate, compare, and construct visual notations. To improve our DSVL's usability and development productivity, we have applied these PoN principles to our visual notation design. Table I lists the PoN principles and presents our visual notation design rules.

In the following paragraphs, we introduce the visual notation design for a suite of visual modeling languages to model endpoint horizontal layers. The data store layer supports the behavior layer, and QoS will be our future work.

B. Signature DSVL Design

To have a concise presentation view of signature model, we have developed a three-level signature DSVL. The top level signature DSVL uses W3C WSDL 1.1 specification [10] as its metamodel and consists of the five entity types defined in the WSDL specification and two relationships to link them together. The middle level operation DSVL is used to specify request and/or response message(s) in an operation (or call service). The bottom level message DSVL defines all element types used in signature modeling; its metamodel is based on W3C XML Schema 1.1 [11]. Using multi-level modeling approach allows the lower level models to be reused by upper level models. The signature DSVL visual notations are presented in Table II.

TABLE II. SIGNATURE DSVL VISUAL NOTATIONS

Visual Symbol	Description [10]
	Service: contains a set of system functions (services) exposed to service consumers through Web-based protocols.
	Port: provides address or connection point to service entity; it has the composite relationship with service entity and associate relationship with binding entity.
	Binding: specifies the interface and defines SOAP binding style and transport; it binds porttype entity to port entity through associate relationship.
	PortType: contains a set of operations a Web service can perform; it has the composite relationship with operation entity and associate relationship with binding entity.
	Operation: is corresponding to a service, and has properties as name and pattern; pattern can be in-only, in-out or out-only.
	Composition relationship: is used to link a main entity to its sub-component entities.
	Association relationship: is used to link two associated entities.
	Message: specifies messages in operation entity; message has properties as element and label, and label can be in or out.
	Complex Element: specifies a complex element in a message; it has properties as element name, type and mandatory.

C. Protocol DSVL Design

We used an Extended Finite State Machine (EFSM) to describe endpoint protocol behaviors. The EFSM metamodel is depicted in Fig. 2. We added one entity type and two entity properties (marked yellow in Fig. 2) to a standard operation-driven finite state machine for enriching our protocol modeling

with dynamic aspects. The entity type is the *InternalEvent*, which allows users to define state transitions triggered by time event. One of the entity properties is the *StateTransitionConstraint* of transition entity, and it is used for specifying either static or dynamic constraints on state transition function. Another one is the *StateTimeProperty* of state entity, which is used to simulate endpoint synchronous process and unsafe operation (not an idempotent operation, which will produce the same results if executed once or multiple times). Table III lists all the visual notations used in our protocol DSVL.

The protocol modeling is only applied to statefull applications. This is because endpoint uses its current state to validate the next coming service. If an endpoint is a stateless application, its protocol modeling will be skipped.

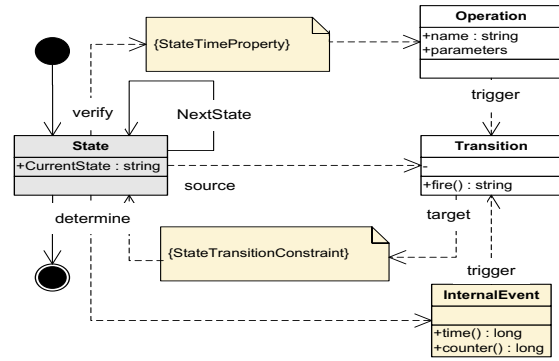


Fig. 2. Protocol DSVL metamodel (EFSM)

TABLE III. PROTOCOL DSVL VISUAL NOTATIONS

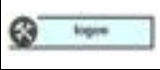





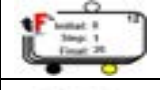
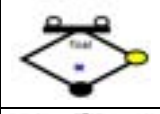
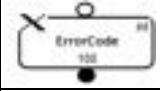

Visual Symbol	Description
	State: presents endpoint state, which normally uses service as its default name; for emulating time related scenarios, the state type has a dialog box for allowing users to define state properties.
	Home state: is a special endpoint state, representing endpoint in active status.
	Idle state: is a special endpoint state, representing endpoint in inactive status.
	Constraint transition relationship: links a <i>from</i> state to a <i>to</i> state for representing a state transition; it has a dialog box for allowing users to define constraint properties.
	Transition relationship: links a <i>from</i> state to a <i>to</i> state for representing a state transition.
	Loop: is used to define a repeat state transition from a <i>from</i> state to a <i>to</i> state.
	Timeout relationship: links two states to represent endpoint state change if no valid service request is received within a defined timeout period.

D. Behavior DSVL Design

Endpoint behavior DSVL was designed based on dataflow programming paradigm [12]. Dataflow programming execution model is represented by a directed graph; the nodes of the graph are data processing units, and the directed arcs between the nodes represent data dependencies. Data flow in each node from its input connector; and the node starts to process and convert the data whenever it has the minimum required parameters available. The node then places its execution results onto output connector for the next nodes in the chain.

To handle complicated business logics, we designed our behavior DSVL as hierarchical tree structure. Each node may contain several sub nodes (or call methods), and each of the sub nodes implements a specific task. The benefits from using the hierarchical structure are two-fold: First, we can reuse some of the nodes, if they perform exactly the same task but are located at different components. Second, it can help us manage diagrammatic complexity problem. At the bottom level, a node consists of some primitive programming constructs for performing operations on data and flow controls for directing execution sequence. We reused the message DSVL of signature DSVL to define data store tables, and a slave table can be defined by specifying the foreign key field data type as undefined.

TABLE IV. BEHAVIOR DSVL VISUAL NOTATIONS

Visual Symbol	Description
	Service node: represents a service provided by an endpoint; it receives request from a SUT, processes it and generates response to the SUT.
	Input and output bars: are used to specify input and output parameters of a service node; all programming constructs will be placed between them to convert input into output; a normal output port (white circle) and an exceptional output port (yellow circle) are on the output bar.
	Data store definition: is used to define data store tables by specifying each table field and field properties.
	Node: is similar to service node, but is used for performing a specific task; a node implementation can either end in successful or failure, this status will decide the next node to be executed in the chain.
	Data store operator: is a primitive construct to retrieve and manipulate data records in data store.
	Evaluator: is a primitive construct to perform an arithmetic operation; the first line is the variable name of the evaluator, the second line lists all variables to be used, and the third line gives the arithmetic formula.
	Loop: is a primitive construct to specify repeated execution of a block of codes for pre-defined times.
	Conditional operator: is a primitive construct to test two input parameters for deciding execution flow; if the testing result is true, the flow will follow the black out port at the bottom; otherwise, the yellow out port at the right will be followed.
	Variable: is a primitive construct to represent a variable with various data types; the variable name is shown at the middle, data type at the upper-right corner and value at the bottom.
	Variable array: is a primitive construct to represent variable array; the array name is shown at the middle, data type at the upper-right corner and array size at the lower-left corner.

At the top level of the node tree structure, discrete service nodes are used to represent the services provided by an endpoint to its SUT. To prevent the data inconsistency between behavior model and signature model, each of the service nodes imports

the request and response parameters from the same endpoint signature model. The service nodes can be collapsed to reduce complexity of the diagram. Table IV lists the main visual notations of our behavior DSVL.

IV. EXAMPLE USAGE¹

A. Business Case

Here, we reuse the banking system of the motivation section and show how a testing endpoint can be modeled by TeeVML. For simplicity, we assume that the core banking system provides only six services to the Internet banking system: session management services *login* and *logout*, a query service *searchaccount*, and three transaction services *deposit*, *withdraw* and *moneytransfer*. We describe the endpoint three interface abstraction layers as below:

Signature – All the services use in-out operation pattern, except for *logout* service that uses in-only pattern. The *login* request has a message ID as mandatory field, and two fields for username and password as optional fields. A SUT can login to the endpoint either in a secured or an insured session, depending on whether the username and password fields are provided or not. All the transaction service requests have a mandatory amount field, which must be equal or greater than zero. All responses contain an optional error code and error message fields for reporting defect types and defect details. For query and transaction services, the response message also includes an optional account balance field to return the account status.

Protocol -- Whenever the endpoint receives a *login* request from its SUT, it transitions from idle state to home state and an interactive session starts. If the endpoint is in secured session, it changes to the service name state whenever receiving a service request. Otherwise, only the query service request is allowed, and its state will change to *searchaccount* state. The endpoint state transition can also be driven by internal time event. If the endpoint current state is timeout, its state will be changed from a service state to the home state or from the home state to the idle state. In addition, the endpoint processes all the services in synchronous mode, and all the transaction services are considered as unsafe services.

Behavior – To start an interactive session, the *login* request must be authenticated against stored user account records, and the request will be rejected if the user ID or the pair of username and password does not match any of those records. For all the transaction and query services, account name and account number are used to search for a bank account in data store, and the account balance will be retrieved. For the *withdraw* and *moneytransfer* services, the retrieved balance must be verified to be equal or greater than the transaction amount. For all the transaction services, we must calculate the new balance amount first, and then write the new balance back to the same bank account record.

B. Signature Modelling

As operation (or call service) definition is the main activity of an endpoint signature modeling, we use the operation *login*

¹ The example application source codes and a recorded demo video are available online: <https://sites.google.com/site/teevmlvhcc/>.

as an example to show how such an operation can be modeled by TeeVML. We start to model the operation by assigning the operation name property as *logon* and selecting in-out from the pattern field drop-down list. Then, operation DSVL is used to specify the both *logon_request* and *logon_response* messages in the operation. The message label is “in” for the request message and “out” for the response message.

Message elements are defined by using message DSVL. The request message contains three elements (or call parameters): *userid*, *username*, and *password*, and they are placed in the message by the ID field in alphabetic order. The *userid* data type is defined as integer and the other two are string, by selecting the corresponding values from the type field drop-down list. An element can either be mandatory or optional by selecting the mandatory field checkbox. *userid* is a mandatory element and *username* and *password* are optional elements. Similarly, we can define the response message of the *logon* operation with three elements: *secure*, *errorcode* and *errormessage*. Fig. 3 illustrates the hierarchical signature model of the core banking system endpoint, including top-level signature model, *logon* operation, and request and response messages.

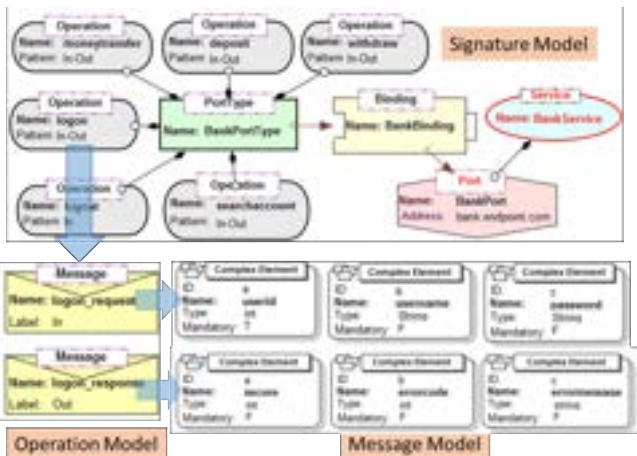


Fig. 3. Example endpoint signature model

C. Protocol Modelling

Endpoint protocol modeling starts from defining its interactive session. A session begins, when the endpoint at idle state receives a *logon* request and changes to home state. This state transition can be represented by using the *logon* transition relationship to link the idle state to the home state. In opposite direction, a session will end, when the endpoint receives a *logout* request at the home state. A session can also be terminated by timeout relationship, linking the home state to the idle state.

Once the endpoint is in a session, it is ready to receive service requests from its SUT. Since a *searchaccount* request will trigger state transition without any constraint, a standard transition relationship can be used to link the home state to *searchaccount* state. However, a constraint transition relationship must be used to link the home state to all transaction states, as transaction services are only allowed in secured sessions. A constraint transition relationship is defined by setting the constraint transition property to a nonnull value

for the username field in *logon* operation. Fig. 4 illustrates the protocol modeling diagram of the banking system endpoint.

To simulate synchronous services, we open the state property dialog box, select synchronous operation checkbox, and provide a number in seconds to the process time field. Similarly, unsafe services are simulated by selecting the unsafe operation checkbox and filling the transmission time field with a number in seconds.

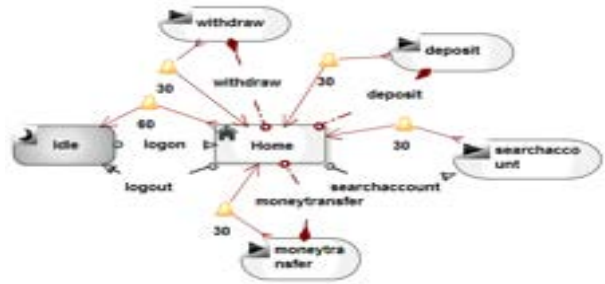


Fig. 4. Example endpoint protocol model

D. Behavior Modeling

We use the *moneytransfer* service node of the banking system endpoint and its sub node *accountinformationretrieve* to explain how the behavior DSVL is used. The *moneytransfer* service node contains three nodes: (1) *accountinformationretrieve* to retrieve the account balance from both “from” and “to” bank accounts, (2) *calculateamount* to calculate the new balances for these two accounts, and (3) *updateaccount* to update the new balance back to persistent data store. Fig. 5a depicts the *moneytransfer* service node structure, execution sequence and dataflow between the input/output bars and nodes.

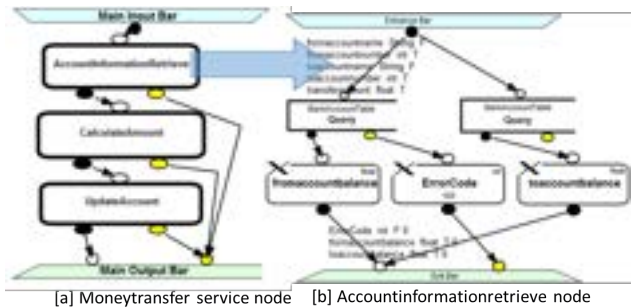


Fig. 5. Example endpoint behavior model

As the *accountinformationretrieve* is the first node to be executed, it will directly take the service node input parameters: *fromaccountname*, *fromaccountnumber*, *toaccountname*, *toaccountnumber* and *transferamount*. The output parameters to the next node are: *fromaccountbalance*, *toaccountbalance* and *ErrorCode*. These input and output parameters are defined when we create the input and output bars. We search “from” account by the *fromaccountname* and *fromaccountnumber* parameters. If the account is found, the account balance will be retrieved and assigned to the variable *fromaccountbalance*. Similarly, “to” account balance will be retrieved by using the *toaccountname* and *toaccountnumber* parameters, and the balance will be assigned to the *toaccountbalance* variable. If the “from” and/or “to” accounts cannot be found, an error occurs and integer

number 100 is assigned to the ErrorCode variable. If the node is executed successfully, both the fromaccountbalance and toaccountbalance variables will be placed on the normal output port. Otherwise, we will put the ErrorCode variable to the exceptional output port. Fig. 5b shows how the *accountinformationretrieve* node is constructed.

V. IMPLEMENTATION

Our Domain-Specific Modeling (DSM) approach in TeeVML is based on the four abstraction-level architecture defined by the OMG Meta Object Facility (MOF) [13]. DSVL development includes language visual notation design and code generator implementation using MetaEdit+ [14].

As part of our TeeVML tool, we have developed two types of code generators for each DSVL by using MetaEdit Report Language (MERL). The first one is to transform models to target source codes: signature model to WSDL 1.1 XML file, and protocol and behavior models to Java classes. The second one is for converting models to SQL scripts for table creation. The signature WSDL file will be further transformed to Web Service engine by using Axis2 code generator utility wsdl2java, acting as our runtime environment.

A complete testing environment consists of a Tomcat application server for hosting the SOAP service provide by endpoint, Axis2 web service engine for SOAP message processing, a protocol class for protocol logic processing, several behavior model classes for generating services to SUT, and MySQL database for storing static and dynamic persistent data. Once all models have been transformed to source codes, we use Apache ant builder to build the endpoint SOAP service, then load the service to the Tomcat application server.

VI. USER EVALUATION

The developed TeeVML was evaluated by a two-phase user survey. In the first phase, we conducted a study with testing experts to examine what features of TeeVML they valued in testing endpoint emulation, and what functionalities such a tool should provide. In the second phase, we evaluated TeeVML's usability by collecting software developers' opinions on their experience with the tool. Specially, we wanted them to compare TeeVML with a third generation language they were familiar with.

A. Experiment Setup

Phase One survey was conducted by interviewing participants. We used a PowerPoint presentation to introduce TeeVML to them, and explained what testing functionalities were required for a testing endpoint and how such an endpoint could be created by use of earlier versions of TeeVML tool. The interview lasted approximately 40 minutes. After the interview, all participants were asked to do an online survey. Since our target audiences for this phase were required to have extensive experience in software testing, we were only able to get 16 participants to take part in this phase. Fig. 6a summarizes Phase One participants IT and software testing experience. As indicated in the charts, most participants were testing experts with solid experience on software development and testing.

Phase Two survey was a three-step process. First, participants watched a recorded video, to introduce TeeVML and show the steps to model a testing endpoint. Second, the participants were assigned a task to model a simple endpoint example. The task was performed by using TeeVML running on a laptop PC. Finally, all participants were asked to do an online survey. The duration for Phase Two was 60 minutes on average. Overall 21 software developers and IT research students took part in the survey. Two participants could not finish the task, due to their personal reasons. Fig. 6b provides Phase Two participants IT background information.

B. Result Analysis and Discussion

We designed total 58 questions for both Phase One and Phase Two to cover various aspects of our TeeVML, and questions types were 5-point Likert Scale (5 to 1 representing strongly agree to strongly disagree) and multi-choice. Due to space limitation, we only selected some of them for this paper evaluation results presentation, and the full result reports are available online². We have counted frequency of participant responses to measure degree of acceptance to a particular question statement. For the 5-point Likert Scale questions, in favour responses encompass the answers of either 5 or 4 for a positive question, or 1 or 2 for a negative question.

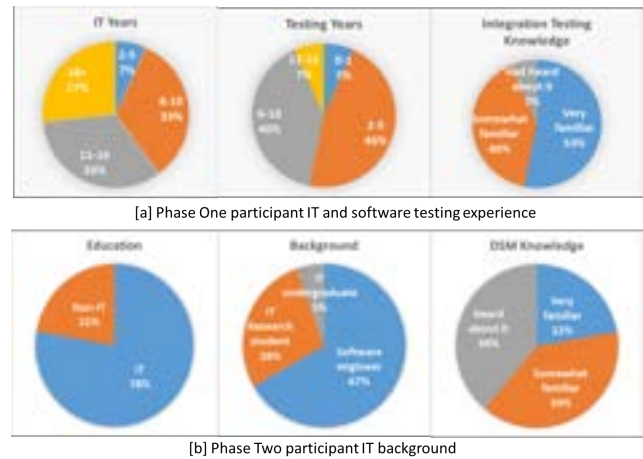


Fig. 6. User survey participants' demographics

1) Phase One Evaluation Results

Table V lists the selected questions and participants' responses. Here we analyse the survey results as follows.

Usefulness – Q8 reflects the overall usefulness of testing endpoint for conducting integration testing. The responses to this question are quite positive with 14 out of 16 participants in favour of usefulness of the tool. We can see that the protocol modeling (Q21) has the highest positive response rate and the non-functional requirement (Q30) the lowest. We believe one of the main reasons why most participants want to have protocol testing is that most applications do not have well-documented protocol specification. Therefore, protocol related defects can only be found by conducting integration testing.

² Refer to the web site: <https://sites.google.com/site/teevmlvhcc/>.

Testing functionalities – Q9 is a multi-choice question for evaluating the usefulness and completeness of functionalities that an endpoint should provide to its SUT. Except for the four features already implemented, we assigned an “Other” item for allowing participants to specify any other useful features, our TeeVML does not support now. Only one participant selected the item “Other”, and suggested to provide performance test under different scenarios. From these responses, we can conclude that most participants are satisfied with the functionalities that TeeVML provides.

TABLE V. PHASE ONE SURVEY RESULTS

No	Statement	Frequency				
		5	4	3	2	1
Q8	In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test.	8	6	0	1	1
Q17	It is useful for an emulated testing environment to provide signature testing functionality to its system under test.	7	7	1	1	0
Q21	It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test.	12	4	0	0	0
Q25	It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test.	6	8	1	1	0
Q30	It is useful for an emulated testing environment to provide non-functional requirement testing features to its system under test.	2	11	3	0	0

No	Question	Statement	Frequency
Q9	What kinds of testing features do you want to see an emulated testing environment provides to system under test for interconnectivity and inter-operability test?	Correctness of message signature	13
		Correctness of interactive protocol	16
		Correctness of interactive behavior	14
		Correctness of non-functional requirement	11
		Other	1
Q13	What are the main motivations for you to use emulated testing environment?	Cost saving on application hardware and software investment	14
		Effort saving on application installation and maintenance	10
		Lack of application knowledge	5
		Early detection of interface defects	15
Q14	What are your main concerns, which could prevent you from using emulated testing environment?	Extra development effort on testing endpoints	6
		Learning a new technology	6
		Inadequate testing functionality	7
		Emulation accuracy	7
		Result reliability	12

Why or Why not use endpoint – Q13 and Q14 are multi-choice questions, and list four reasons and five concerns why or why not users want to use testing endpoints. Surprisingly, the top reason for users to use endpoints is the early detection of interface errors, rather than savings on investment and development effort. Early interface defects detection is particularly important, when an application is developed by a third party and environment systems are completely inaccessible. Q14 reflects most participants’ concern on result reliability. We believe the main reason is that in new endpoint development process endpoints are modeled rather than coded. Therefore, it is important to reliably model emulated testing environments.

2) Phase Two Evaluation Result

For Phase Two, we evaluate the overall usability of TeeVML using Software Usability Scale (SUS) [15]. Table VI presents the SUS survey results by frequencies. We have received quite positive responses from the survey participants, with average 16.2 in favour. Particularly, the Q17 has received in favour response from all participants, followed by Q12, Q14 and Q16. The lowest score is Q15 that has less than half (8 participants) in favour. The assigned task was actually modeling related and a certain level of modeling skill was required. However, the survey participants were not mostly experts in domain specific modeling (refer to Fig. 6b). So, some of them might have needed support for modeling related techniques.

TABLE VI. FREQUENCY TABLE OF SOFTWARE USABILITY SCALE

No	Statement	Frequency				
		5	4	3	2	1
Q12	You would like to use the tool in your future project.	7	11	1	0	0
Q13	You found the tool unnecessarily complex.	0	1	2	12	4
Q14	You found the tool was easy to use.	8	10	1	0	0
Q15	You would need support to be able to use the tool.	0	2	9	8	0
Q16	You found the various features of the tool were well integrated.	8	10	0	1	0
Q17	You found there was too much inconsistency in the tool.	0	0	0	11	8
Q18	You would image that most people would learn to use the tool very quickly.	5	11	1	1	0
Q19	You found the tool very cumbersome to use.	0	0	2	10	7
Q20	You felt very confident using the tool.	4	13	2	0	0
Q21	You needed to learn a lot of things before you could get going with the tool.	0	1	3	8	7

TABLE VII. INTERFACE LAYER USABILITY QUESTIONS AND RESPONSES

No	Question	Frequency				
		5	4	3	2	1
Q27	Endpoint signature is easily modeled by the tool.	9	9	0	0	1
Q29	It is easy to make changes to message signature model.	13	6	0	0	0
Q30	It is easy to make errors or mistakes during message signature definition.	0	3	7	5	4
Q31	It is capable of defining all types of message signatures you have seen.	2	11	6	0	0
Q33	Endpoint protocol is easily modeled by the tool.	12	7	0	0	0
Q35	It is easy to make changes to interactive protocol model.	13	6	0	0	0
Q36	It is easy to make errors or mistakes during interactive protocol definition.	1	1	5	6	6
Q37	It is capable of defining all interactive protocol scenarios you have seen.	4	8	6	1	0
Q39	Endpoint interactive behavior is easily modeled by the tool.	3	14	1	1	0
Q41	It is easy to make changes to interactive behavior model.	10	9	0	0	0
Q42	It is easy to make errors or mistakes during interactive behavior definition.	1	1	11	6	0
Q43	The tool has sufficient expressive power for creating behavior model with accurate outputs.	1	9	8	1	0

Table VII presents Phase Two survey questions and responses from three interface layers: signature, protocol and behavior, and four usability dimensions of each layer: ease of use, maintainability, error prevention and completeness. Fig. 7

summarizes the in favour responses for each layer and usability dimension from Table VII.

From the layers' viewpoint (refer to Fig. 7a), protocol DSVL has the highest usability and behavior the lowest. This result is in coincidence with what we have expected. Endpoint protocol modeling is simple and easy, and only four relationship types are used to specify various state transitions. In contrast, behavior modeling must deal with complicated logic processing, involving data manipulation, flow control, data store access, etc.

For the usability dimension (refer to Fig. 7b), maintainability has received in favour response from all participants, and is followed by ease of use. High maintainability is one of the key motivations for us to select a DSVL approach, since any changes to endpoint can be done by modifying models only and engaging in coding is not required. More than half of participants were not satisfied with the error prevention mechanism provided by TeeVML. Although TeeVML supports most DSVL specific error prevention mechanisms, it does not currently provide comprehensive error and type checking.



Fig. 7. Summary of in favour responses for different layers and dimensions

VII. RELATED WORK

There are a wide variety of domain-specific languages. They can be widely used languages for a specific technical domain, such as HTML for web pages, SQL for relational databases, and WebDSL for web applications [16]. Or, they can narrowly focus on a specific business domain, such as MaramaEML for business process modeling [17], SDL for supporting statistical survey process [18], and LabVIEW for electronic circuit testing design [19]. In contrast, we use a set of domain-specific visual modeling languages tailored to modeling signature, protocol and behavior layers of endpoints.

A testing endpoint is developed from its external behavior, communicating with other software components. Han first proposed a rich interface definition framework [7] with layers: signature, configurations, semantics, constraints, and a quality aspect across all these layers. Han's framework defines how to select and reuse a software component, not just based on static component signature, but also on other runtime aspects as well. From a service viewpoint, Beugnard et al. defined a four-level software component contract template with increasingly negotiable properties along with the levels [20]. Our approach on the other hand, focuses on how a request is to be processed by an endpoint in a layered manner.

For the protocol modeling, some researchers used a finite state machine [21, 22] or a formal protocol specification [23, 24] to validate message sequence for different endpoint states. However, Wehrheim et al. argued that the use of service name

alone might not be sufficient to trigger a state transition for a realistic endpoint [25]. To deal with the so-called incomplete protocol specification, [25] developed an EFSM-based protocol modelling calculus for specifying service parameters and return values as runtime constraints. Although, various notions for protocol specification have been suggested, there are still some issues to be solved. One is lack of concrete implementation solutions to capture endpoint runtime aspects. Another one is textual form they used for writing state transition rules, and this will make protocol modeling difficult.

Software components interface behaviors can be modeled either externally or internally. Software behavioral interface specification [26] and programming from specification [27] are the external approaches, they model interactive behaviors by defining pre/post conditions to bind both service consumer and service provider. As internal approaches, Business Process Model and Notation (BPMN) [28] and DataFlow Programming (DFP) [12] provide graphical notations for specifying internal data processes and flow controls. In general, external approaches and BPMN require extensive modeling and programming work. While, DFP languages are ease of use with user-friendly interface. But, they are less expressive and only suitable for a specific domain. In contrast to these approaches, our behavior DSVL is ease of use by dragging-and-dropping visual symbols. For handling complicated business logics, hierarchical nodes tree structure is adopted.

UML is a widely used general purpose modeling language, focusing on the definition of system static and dynamic behaviors. Specifically related to our work, UML provides: (1) a testing profile to provide a generic extension mechanism for the automation of test generation processes [6], (2) state charts to simulate finite-state automaton [29], and (3) activity diagrams to graphically represent workflows of stepwise activities and actions [30]. Two main problems with using UML to define new modelling languages [31] are that it is usually hard to remove parts of UML that are not relevant or need to be restricted in a specialized language; and all the diagram types have restrictions based on the UML semantics.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a suite of domain-specific visual languages for testing environment emulation. The language consists of three DSVLs for each interface layer. An endpoint is modeled using these three abstraction levels. By this layered approach, our TeeVML supports partial endpoint development, where a testing endpoint may have only one or two of these layers to meet SUT testing requirement. We have conducted a survey to evaluate the tool's functionality and usability. The survey results demonstrated acceptance of the tool among software testing experts and developers and further improvement areas, such as error prevention.

A fully functional testing endpoint must include testing Quality of Service (QoS) provided to SUT. The QoS DSVL should be able to model performance, reliability, security and other non-functional attributes. We are investigating the benefits of building another DSVL specifically for model syntax checking before transforming models to target sources as our current future work focus.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge support for this research by an Australian Post-graduate Award and an Australian Research Council Discovery Projects grant.

REFERENCES

- [1] P. B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," *IEEE Transactions on Software Engineering*, vol. 13, pp. 77-87, 1987.
- [2] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," presented at the In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Canada, 2004.
- [3] J. Yu, J. Han, J.-G. Schneider, C. Hine, and S. Versteeg, "A virtual deployment testing environment for enterprise software systems," presented at the Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, Italy, 2012.
- [4] M. Du, J.-G. Schneider, C. Hine, J. Grundy, and S. Versteeg, "Generating service models by trace subsequence substitution," presented at the Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, Canada, 2013.
- [5] C. Hine, J.-G. Schneider, J. Han, and S. Versteeg, "Scalable emulation of enterprise systems," in *Software Engineering Conference*, Australian, 2009, pp. 142-151.
- [6] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, "The UML 2.0 testing profile and its relation to TTCN-3," in *Testing of Communicating Systems*, ed: Springer, 2003, pp. 79-94.
- [7] J. Han, "Rich Interface Specification for Software Components," Peninsula School of Computing and Information Technology Monash University, McMahons Road Frankston, Australia, 2000.
- [8] D. L. Moody, "The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering," *Software Engineering*, *IEEE Transactions on*, vol. 35, pp. 756-779, 2009.
- [9] J. H. Larkin and H. A. Simon, "Why a Diagram is (Sometimes) Worth Ten Thousand Words," *Cognitive Science*, vol. 11, pp. 65-100, 1987.
- [10] E. Christensen, F. Curbera, and G. Meredith, "Web Services Description Language (WSDL) 1.1. W3C," Note 15, 2001, www.w3.org/TR/wsdl, 2001.
- [11] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML schema part 1: structures second edition," ed: W3C Recommendation, 2004.
- [12] T. B. Sousa, "Dataflow Programming Concept, Languages and Applications," in *Doctoral Symposium on Informatics Engineering*, 2012.
- [13] OMG, "Meta Object Facility (MOF) Specification," ed: The Object Management Group, 2000.
- [14] S. Kelly, "Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM," in *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development*, 2004.
- [15] J. Brooke, "SUS-A quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, pp. 4-7, 1996.
- [16] E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," in *Generative and Transformational Techniques in Software Engineering II*. vol. 5235, R. Lämmel, J. Visser, and J. Saraiva, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 291-373.
- [17] L. Li, J. Grundy, and J. Hosking, "A visual language and environment for enterprise system modelling and automation," *Journal of Visual Languages & Computing*, vol. 25, pp. 253-277, 8// 2014.
- [18] C. H. Kim, J. Grundy, and J. Hosking, "A suite of visual languages for model-driven development of statistical surveys and services," *Journal of Visual Languages & Computing*, vol. 26, pp. 99-125, 2015.
- [19] J. Travis and J. Kring, *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition)*. Upper Saddle River, NJ, USA Prentice Hall PTR, 2006.
- [20] A. Beugnard, J.-M. J. I. Plouzeau, and D. Watkins, "Making Components Contract Aware," *Computer*, vol. 32, pp. 38-45, 1999.
- [21] L. De Alfaro and T. A. Henzinger, "Interface automata," *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 109-120, 2001.
- [22] U. Endriss, N. Maudet, F. Sadri, and F. Toni, "Protocol conformance for logic-based agents," in *IJCAI*, 2003, pp. 679-684.
- [23] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols," in *Technology of Object-Oriented Languages and Systems, TOOLS 30 Proceedings*, 1999, pp. 387-398.
- [24] Y. Jin and J. Han, "Specifying Interaction Constraints of Software Components for Better Understandability and Interoperability," in *COTS-Based Software Systems*. vol. 3412, X. Franch and D. Port, Eds., ed: Springer Berlin Heidelberg, 2005, pp. 54-64.
- [25] H. Wehrheim and R. H. Reussner, "Towards more realistic component protocol modelling with finite state machines," *UNU-IIST*, p. 27, 2006.
- [26] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. MULLER, and M. Parkinson, "Behavioral interface specification languages," *ACM Comput. Surv.*, vol. 44, pp. 1-58, 2012.
- [27] C. Morgan, *Programming from specifications*: Prentice-Hall, Inc., 1990.
- [28] M. von Rosing, S. White, F. Cummins, and H. de Man, "Business Process Model and Notation—BPMN," 2015.
- [29] S. J. Zhang and Y. Liu, "An Automatic Approach to Model Checking UML State Machines," in *Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, 2010 Fourth International Conference on, 2010, pp. 1-6.
- [30] M. Dumas and A. H. Ter Hofstede, "UML activity diagrams as a workflow specification language," in *« UML » 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, ed: Springer, 2001, pp. 76-90.
- [31] A. Abouzahra, J. Bézivin, M. D. Del Fabro, and F. Jouault, "A practical approach to bridging domain specific languages with UML profiles," in *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, 2005.

5

Software Process Management with DSLs and MDE

5.1 Serendipity: integrated environment support for process modelling, enactment and work coordination

Grundy, J.C. and Hosking, J.G. Serendipity: integrated environment support for process modelling, enactment and work coordination, *Automated Software Engineering*, Vol. 5, No. 1, January 1998, Kluwer Academic Publishers, pp. 27-60.

DOI: [10.1023/A:1008606308460](https://doi.org/10.1023/A:1008606308460)

Abstract: Large cooperative work systems require work coordination, context awareness and process modelling and enactment mechanisms to be effective. Support for process modelling and work coordination in such systems also needs to support informal aspects of work which are difficult to codify. Computer-Supported Cooperative Work (CSCW) facilities, such as inter-person communication and collaborative editing, also need to be well-integrated into both process-modelling tools and tools used to perform work. Serendipity is an environment which provides high-level, visual process modelling and event-handling languages, and diverse CSCW capabilities, and which can be integrated with a range of tools to coordinate cooperative work. This paper describes Serendipity's visual languages, support environment, architecture, and implementation, together with experience using the environment and integrating it with other environments.

My contribution: Developed initial ideas for the research, did majority of tool design, implemented and evaluated tool, wrote majority of the paper, investigator for funding for the project from FRST

Serendipity: integrated environment support for process modelling, enactment and work coordination

JOHN C. GRUNDY¹ AND JOHN G. HOSKING²

¹*Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand
Email: jgrundy@cs.waikato.ac.nz;*

²*Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
Email: john@cs.auckland.ac.nz*

Abstract. Large cooperative work systems require work coordination, context awareness and process modelling and enactment mechanisms to be effective. Support for process modelling and work coordination in such systems also needs to support informal aspects of work which are difficult to codify. Computer-Supported Cooperative Work (CSCW) facilities, such as inter-person communication and collaborative editing, also need to be well-integrated into both process-modelling tools and tools used to perform work. Serendipity is an environment which provides high-level, visual process modelling and event-handling languages, and diverse CSCW capabilities, and which can be integrated with a range of tools to coordinate cooperative work. This paper describes Serendipity's visual languages, support environment, architecture, and implementation, together with experience using the environment and integrating it with other environments.

Keywords. Process modelling, process enactment, process-centered environments, work coordination, environment integration

1. Introduction

Most computerised or semi-computerised work systems have evolved informal or semi-formal process models. These attempt to describe the use of different tools on a project, the interchange and modification of work artefacts by tools and workers, and the flow of control and/or data. Workflow Management Systems (WFMS) and Process-Centred Environments (PCEs) are examples of tools developed to assist in the construction, use and evolution of formalised process models. While many of these systems support modelling and enacting work processes well, they have some deficiencies. Some use low-level, textual process models which are difficult for end-users to understand and modify (Swenson, 1993; Bogia and Kaplan, 1995), some lack support for inter-person communication (Di Nitto et al., 1995; Ben-Shaul and Kaiser, 1996; Tolone et al., 1995), and many provide either inflexible process models, which inadequately handle exceptions and are difficult to change while in use (Swenson, 1994; Tolone et al., 1995), or do not adequately support the more informal aspects of cooperative work (Kaplan et al., 1996; Tolone et al., 1995).

Computer-Supported Cooperative Work (CSCW) systems provide communication and cooperative editing tools to support collaborative work. These include synchronous editors and tools, such as shared workspaces, telepointers and video conferencing, and asynchronous communication, such as email and version control systems (Ellis et al., 1991). However, many of these systems focus on low-level collaborative editing and/or person-to-person communication issues, and lack support for general work process modelling (Krishnamurthy and Hill, 1995). Some systems, such as ConversationBuilder (Kaplan et al., 1992a), SPADE/ImagineDesk (Di Nitto and Fuggetta, 1995), and Oz (Ben-Shaul and Kaiser, 1994a), attempt to bridge the gap between CSCW tools and process modelling tools but with limited success. Others, such as wOrlds (Bogia and Kaplan, 1995), provide facilities for supporting the informal aspects of work but do not adequately support codified, cooperative process models (Kaplan et al., 1996). Most existing CSCW, WFMS and PCEs tend not to be well integrated with each other, nor with work

tools, such as document editors, CASE tools and programming environments (Tolone et al., 1995, Ben-Shaul and Kaiser, 1996). This usually means custom-built systems must be developed which exhibit CSCW behaviour, or existing tools modified to permit (often loose) integration with process modelling tools.

We describe Serendipity, which integrates process modelling and cooperative work support for large collaborative systems. Serendipity provides a novel graphical process modelling language which can be used to represent both general process models and plans for and histories of work. Another graphical language specifies event handling for process model enactment and work artefact modification events. We focus on the design of these languages and a supporting environment for them, together with our experiences in integrating this environment with other systems, including both CSCW tools and tools for performing work, such as CASE, programming environment, and office automation tools. Serendipity provides a “work context” for these tools, and supports high-level work context awareness and work coordination for them. Descriptions of work artefact changes from integrated tools are annotated with work context information and stored by the current enacted process stage, along with process enactments, forming work and enactment histories. Serendipity is integrated with existing tools via an event passing system, necessitating no modifications to these tools. Serendipity has been used for collaborative process modelling and work planning, process improvement, method engineering and office automation.

2. Problem Domain: Large Cooperative Work Systems

Tools that effectively support collaborative work in large problem domains, where several people, tools and many work artefacts are involved, such as software development or office automation, have a number of important requirements:

- Support for modelling work processes (Barghouti, 1992; Swenson et al., 1994; Tolone et al., 1995). This allows cooperating people to more readily plan and coordinate their work on large problems. Approaches to providing such process modelling range from precise, formal languages (Barghouti, 1992; Ben-Shaul and Kaiser, 1994a; Bandinelli, et al., 1994) to more high-level, graphical workflow languages (Medina-Mora et al., 1992; Swenson et al., 1994; Baldi et al., 1994). Process models should ideally allow work processes to be enforced or used as guidance, and should be readily understandable and modifiable by users.
- The ability to handle arbitrary events relating to process model state changes, work artefact updates or events in work tools (Bogia and Kaplan, 1995; Grundy et al., 1995a). When the state of a process model or work artefact changes, interested collaborators may need to be informed, or automatic responses carried out (Ben-Shaul and Kaiser, 1994a; Bogia and Kaplan, 1995).
- Effective inter-person communication and collaborative editing support is needed (Di Nitto and Fuggetta, 1995; Ben-Shaul and Kaiser, 1996). With process models providing a high level of “work context” information, keeping collaborators aware of each others’ work is necessary (Bogia, 1995). Allowing people to intermittently join or leave cooperative work sessions, or to review the reasons for artefact changes, requires annotated histories of work for process stages.
- Tools for performing work need to be well-integrated with the tools for communicating and editing work artefacts (CSCW tools) and the process modelling tools (WFMS and PCEs) (Di Nitto and Fuggetta, 1995; Ben-Shaul and Kaiser, 1996, Kaplan et al., 1996). This should allow: CSCW tools to be utilised to discuss and annotate both work and process model artefacts; the process modelling tools to control CSCW and work tool behaviour; work performed with the work tools to be readily coordinated by the process modelling tools.

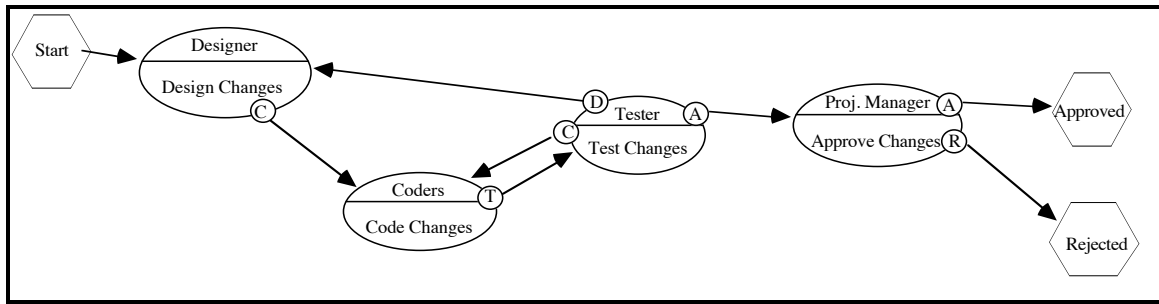


Figure 1. An example VPL process model.

As an example of a process modelling language and environment, consider Swenson's Visual Planning Language (Swenson, 1993), and its embodiment in support environments Regatta (Swenson et al., 1994) and TeamFLOW (TeamWARE, 1996). Figure 1 shows a VPL model describing a process to modify a software system. Ovals are process stages, hexagons start/stop states, and flows between stages represent finishing states of one stage which enact the linked stages.

VPL has many advantages over comparable process modelling languages in that it is concise and versatile. It can be used to both model processes and plan work on specific projects, and Regatta and TeamFLOW allow its process models to be enacted by multiple collaborating users. However, VPL does not adequately model work artefacts or tools, using only simple textual attributes of process stages to indicate this information. It does not support the handling or arbitrary events from related process stages nor tools and artefact updates. The integration of its supporting environments with tools for performing work is limited. For example, TeamFLOW is unable to handle events from work tools or directly control their usage, other than invoking them, when appropriate, from process model stages.

Some work has been done on integrating PCE/WFMS, CSCW and work tools. Examples include ConversationBuilder (Kaplan et al., 1992a), wOrlds (Tolone et al., 1995), SPADE and ImagineDesk (Di Nitto and Fuggetta, 1995), Marvel and ProcessWEAVER (Heineman and Kaiser, 1995), and Oz (Ben-Shaul et al., 1994b; Ben-Shaul and Kaiser, 1996; Ben-Shaul and Kaiser, 1994a). These approaches have produced useful environments with some integration of PCEs/WFMSs, CSCW tools and tools used to perform work. However, this work has so far not produced the "ideal" integration of these technologies, due to problems with adequately integrating disparate tools (Tolone et al., 1995; Valetto and Kaiser, 1995; Di Nitto and Fuggetta, 1995), or with limited scope of the software development and/or workflow tools used (Kaplan et al., 1996, Di Nitto and Fuggetta, 1995).

Serendipity is our approach to solving these problems. It extends VPL to more completely describe work processes, and adds a new visual language for specifying event handling. The next two sections describe Serendipity's visual languages, and are followed by a description of the support environment. Section 6 describes integration of Serendipity with other tools, and is followed by a description of the architecture and implementation. Section 8 describes experience using Serendipity, compares it with other languages and systems, and discusses current and future work.

3. High-level Process Modelling and Work Planning

3.1. EXTENDED VISUAL PLANNING LANGUAGE

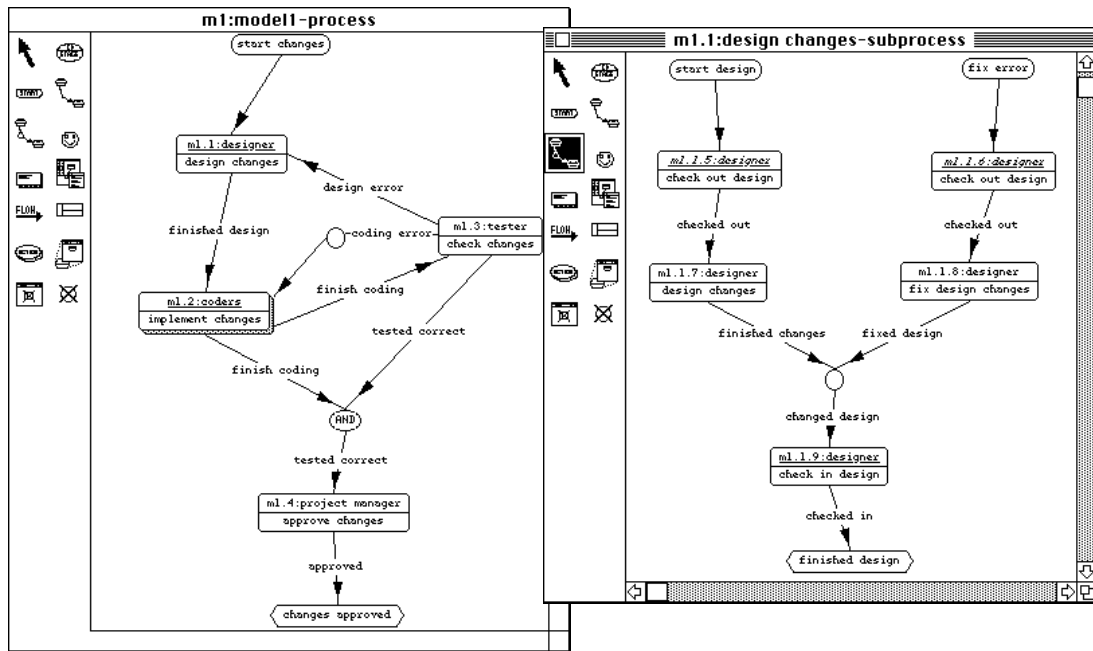


Figure 2. A simple software process model and subprocess model in Serendipity.

We have developed the Extended Visual Planning Language (EVPL), which preserves the notion of simple “work plans” from VPL, but adds capabilities to support general process modelling. EVPL can thus be used to both model generic, reusable work processes, and to model and record work plans and histories for a particular project. New notational components in EVPL include: identifiers for process stages, which allow stages to be more readily identified and referenced by other process models; role, artefact and tool representations, so these aspects of the work context for a process stage can be specified; “usage” connections between process stages and role, artefact and tool representations; and various annotations on usage connections to indicate how these other parts of a work context are utilised by process stages. Window “m1:model1-process” in Figure 2 uses EVPL to describe a simple software process model for updating a software system. Figure 3 summarises the basic EVPL modelling capabilities.

Notational Symbol	Example	Description
		An EVPL Process stage. Stages have a unique ID, a role (person or people who perform the stage tasks), and name. The ID is user-defined and used to uniquely identify stages when they appear in multiple views of a process model.
		An EVPL stage the tasks of which are performed by multiple people (i.e. several people are assigned to the role).
		Start stage for an EVPL diagram. When a process is first enacted, an enactment flow event will flow in from one of its start stages and onto the connected process stage(s).
		Stop stage for an EVPL diagram. When a finishing stage flows into a stop stage, the EVPL process has completed.
		An enactment event flow from one stage to another. The label on the enactment flow is usually an indication of the finishing state of the process stage from which the flow is from. In the example, if stage 1.1:Design Changes finishes in state “finished designing”, stage 1.2:Modify Code is enacted.

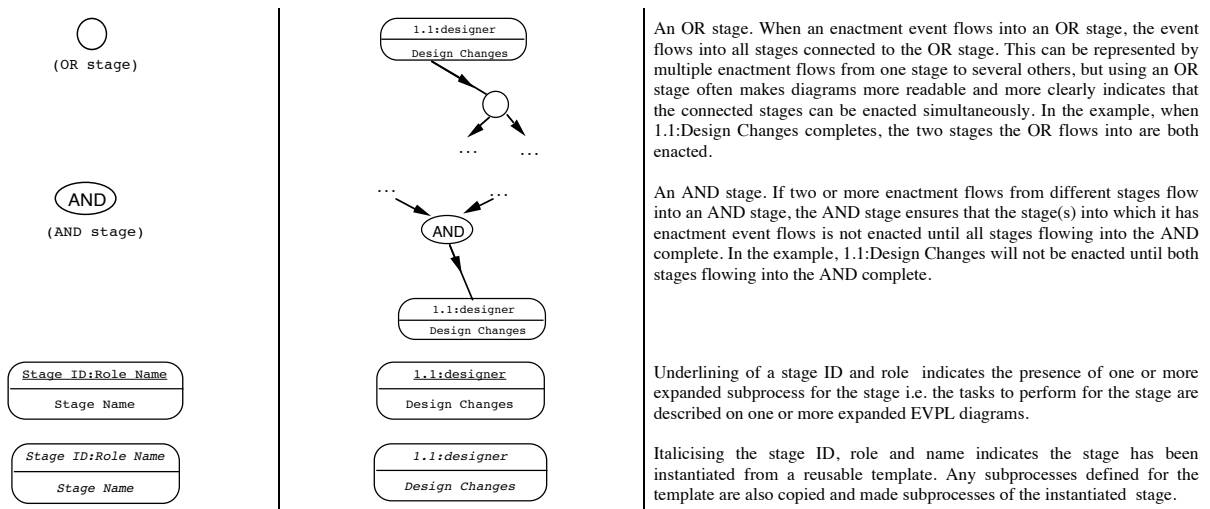


Figure 3. Basic modelling capabilities of EVPL.

The Figure 2 example would normally be part of a larger process, such as the ISPW-6 software process example (Kellner et al., 1990). Several stages are shown, each describing part of the overall process. Process stages “m1.1: design changes” and “m1.2: implement changes” have subprocess models. The expanded subprocess model for “m1.1: design changes” is shown in the window on the right. Stages m1.5 and m1.6 in the subprocess model have been instantiated from templates.

The *AND stage* between m1.2, m1.3 and m1.4, implies stages m1.2 and m1.3 must both finish in the given finishing states before m1.4 is enacted. This does not necessarily mean all subprocesses of m1.2 or m1.3 need be finished, as they may have parallel flows which terminate with different finishing states at differing times. However, at least one flow must terminate m1.2 and m1.3 in the specified finishing states for m1.4 to be enacted. The *OR stage* (unlabelled circle) connecting m1.1.7, m1.1.8 and m.1.9, indicates m1.1.9 is enacted when either m1.1.7 or m1.1.8 finish.

To use this process model, stages in the model are *enacted*. When a stage completes in a given state, event flows with that state name (or no name) activate to enact linked stages; enacted stages and the enactment event flows causing their enactment are highlighted. Enactments of each stage are recorded, as are the work artefact changes made while a stage is the *current enacted stage* for a user (i.e. that user’s work context). Section 5 further describes process enactment in Serendipity.

Empty, or leaf, stages have no definition of their work process. In addition, some process model data, such as roles, may be abstract, requiring specification for a particular project the process is used on. Users working on a particular project can define work plans for leaf stages, and extend models to more precisely specify role, artefact and tool data. For example, the process model above indicates work plans are defined for each “coder” for stage m1.2. The people filling these “coder” roles can define or modify these work plans or have them defined by some other role (perhaps the “designer” or “project manager”).

Figure 4 shows the EVPL work plan definition for the “m1.2: implement changes” stage for coder “john”. When this stage is enacted with “start coding”, “john” is to add a relational database table “address”, modify a table “customer”, and then modify any forms and reports affected by these schema changes. If the stage is enacted with “fix code”, “john” must locate the error and fix the schema, forms and reports. Different work plans can be defined for other people filling the “coder” role. This illustrates the versatility of EVPL for both defining work process models and planning the actual work to be done on a particular project.

Note that there is a mapping from the “finished design” and “coding error” finishing states of m1.1 and m.1.3 to the “start coding” and “fix code” starting state names of the subprocess model. This mapping is

specified in a dialog when adding each enactment event flow. Stages can be enacted in the same start state by multiple event flows, such as stage m1.2.6 in Figure 4.

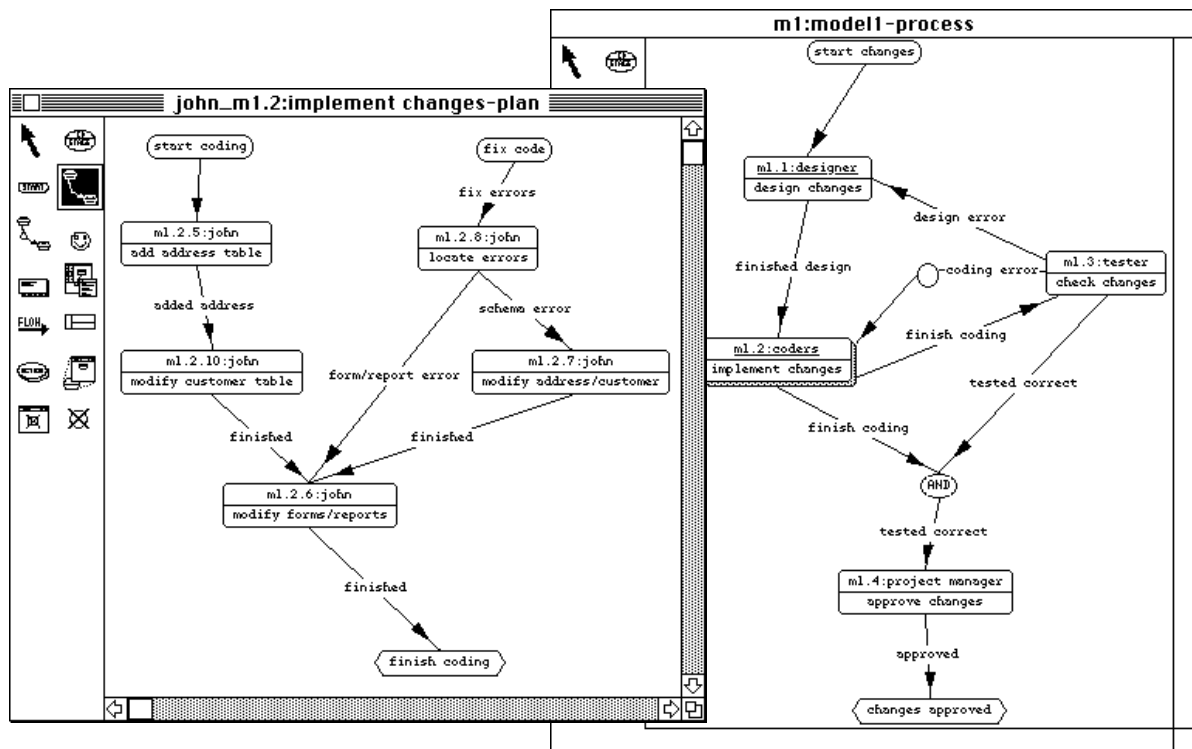






Figure 4. A work plan for coder “john”.

4.2. DESCRIBING PROCESS STAGE WORK CONTEXT INFORMATION

EVPL also extends VPL to permit capture of *work context* information in the form of tools and artefact used in each stage, and coordination and communication needed between stage roles (Figure 5). Some, but not all, of these additional modelling capabilities are found in process modelling languages developed by other researchers (for examp, E³ PML (Baldi et al., 1994)).

Figure 6 shows a different perspective (view) of the process model of Figure 2. This retains the process stages, but not the enactment event flows. Instead, usage flows describe the tools, artefacts and roles that the stages use. For example, “m1.1:design changes” uses the “ER Modeller” tool and the “ER design” artefact. The usage flow between “ER design” and “ER Modeller” specifies the latter is used to modify the former. The usage connections to the “designer” role from m1.1 and m1.3 indicate coordination between the m1.3 role (“tester”) and the m1.1 role (“designer”), in this case the roles must communicate informally (“talks with”). Annotations on the usage flows indicate, for example, that the designer must use the ER modeller to design changes (√), the ER design artefact is updated by m1.1:design changes (U), the changes list artefact is created and/or updated by m1.1:design changes (CU), m1.2:implement changes accesses (but doesn’t update) the ER design (A), and process stages m1.2 and m1.3 cannot be enacted at the same time as process stage m1.4 (¬).

Notational Symbol	Example	Description
 artefact	 class	These describe the kinds of work artefacts used by process stages. Artefacts can represent a general class of work artefact (e.g. “classes” in an OO system), or specific instances of artefacts (e.g. “class ‘window’”).
 ToolName	 ER Modeller	Tools are used to modify or view work artefacts. If a meta-process model is being defined, tools may include the Serendipity environment itself, and artefacts may include process model components.

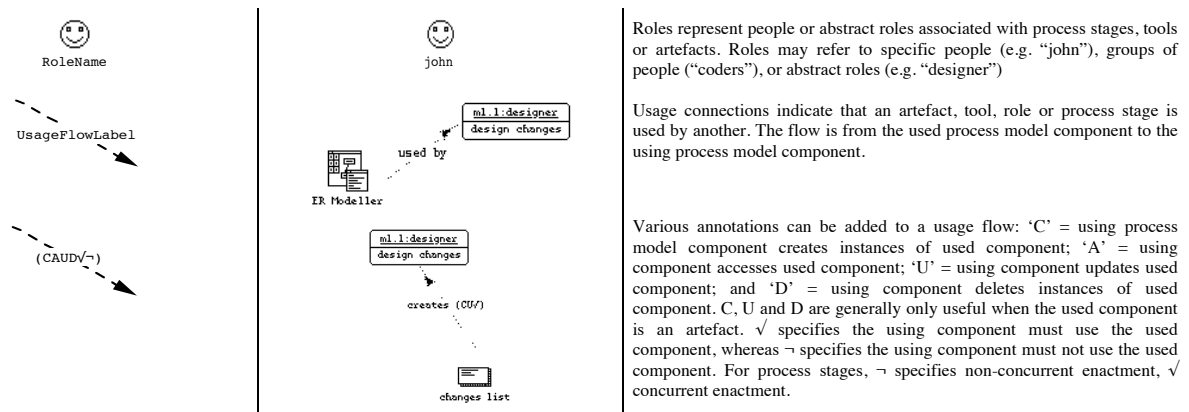


Figure 5. Extra modelling capabilities of EVPL to describe work contexts.

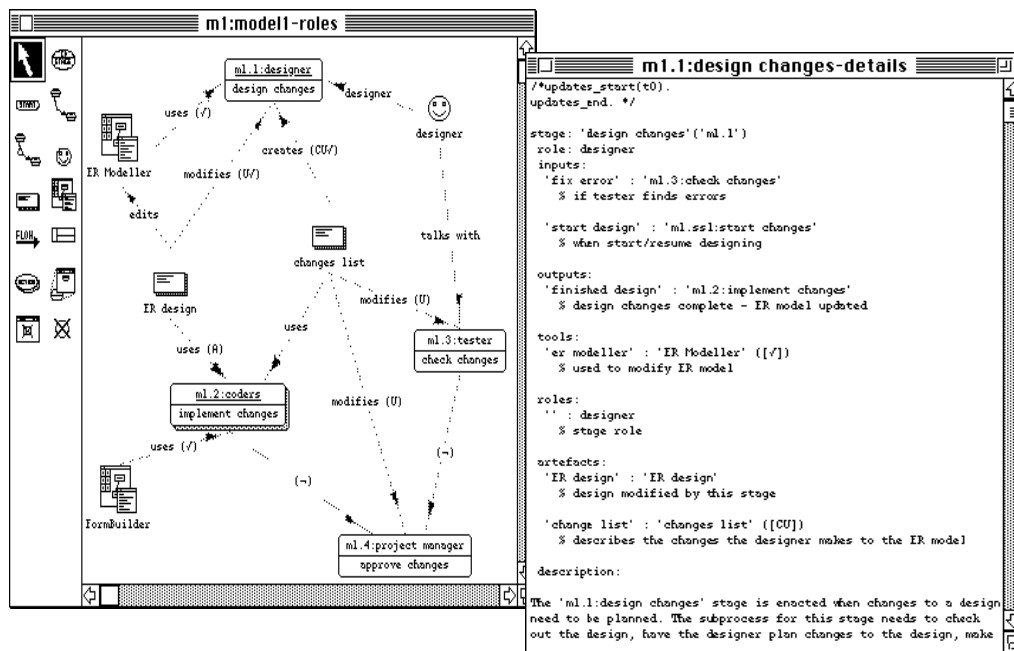


Figure 6. A data, tool and role-oriented perspective of the first process model (left) and a textual stage view (right).

3.3. TEXTUAL PROCESS INFORMATION

The graphical process model, work plan and work context views described above, can be supplemented using textual representations of process stage, artefact, tool and role information. An example is given in Figure 6. Extra information which can be specified includes: user-defined attributes for any kind of process model component (for example, a "percent_complete" value for a process stage); user-defined comments about process model component information; and bindings of abstract role to concrete role (e.g. that "john" will fill the "designer" role).

4. Defining Event-Handling Filters and Actions

The process models described in the previous section are fairly static. Stage enactment flows are described, which indicate the flow of enactment events between stages. Often, however, other types of event handling are required, for example to specify: dependencies between process stages which belong to different process model views; the handling of artefact update and/or tool events; automatically-applied

rules or constraints on process models, driven by event triggers; and the automatic invocation of inter-person communication tools.

Few graphical process modelling languages support such capabilities. Most typically use some form of textual specification of process model stage attributes to specify limited forms of event handling, or utilise complex rule-based languages. Neither of these approaches capture and represent graphically the kinds of events nor how they are handled. To this end we developed the Visual Event Processing Language (VEPL), to permit visual specification of arbitrary event handling and event-triggered process model rules.

4.1. BASIC EVENT-HANDLING

The basic VEPL constructs are *filters* and *actions*, which receive *events* from stages, artefacts, tools or roles, or other filters and actions. Filters match received events against user-specified criteria, passing them onto connected filters and actions if the match succeeds. When actions receive an event they carry out one or more operations in response to the event. These operations update information and/or generate new events, which may be detected and acted upon by other filters and actions. VEPL is fully integrated with EVPL and thus provides a graphical, high-level specification of event handling for EVPL process models and work plans.

Figure 7 shows the main components of VEPL. Some of these, such as enactment event flows and usage connections, are derived from EVPL, but have different semantics when used with VEPL filters and actions.

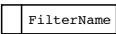
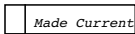
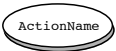

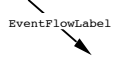
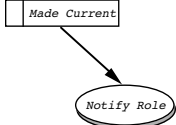
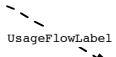
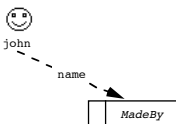
Notational Symbol	Example	Description
		A filter definition. Filters receive events (from process stages, artefacts, tools, roles, other filters or actions) and if the event matches the defined selection criteria for the filter, the event is passed onto the connected filters and/or actions. A filter or action reused from a template filter/action definition has its name in italics.
		An action definition. Actions receive events (from process stages, artefacts, tools, roles, filters or other actions) and respond to the event by performing some action (which often generates other events). Actions can pass on events to other filters and/or actions.
		An event flow into/from a filter or action. Events may be process stage enactment events, artefact update events, tool events, some event caused by a role (i.e. user), or an event generated by an action. For example, if Made Current decides an enactment event flowing into it means a process stage has been made the current enacted stage, then the Notify Role action is invoked to notify another user about this event.
		Usage flow into a filter or action. These specify parameters of the filter/action. For example, the MadeBy filter is parameterised by a role name which it uses to decide whether some event was caused by a particular role. In the example, that role name is instantiated to "john" by the usage connection to the role process model component.

Figure 7. Basic modelling capabilities of VEPL.

Figure 8 shows a simple event-handling view which extends the process model from Figure 2, illustrating use of VEPL for inter-stage work coordination. In this example, testing of software (stage m1.3) can be carried out while further design and/or coding takes place. To coordinate the people associated with the “designer”, “coders” and “tester” roles, Figure 8 specifies when notifications are sent to coders that testing has been completed or is in progress.

The lefthand event flow from m1.3 specifies that all enactment events on m1.3 should flow into the filter “Made Current”. This checks if the received event shows the stage has been made the user's current enacted stage (i.e. the user is now working on this stage) and, if so, the event flows into the OR stage and on to “Not Completed”; otherwise event propagation stops. “Not Completed” is parameterised by a stage

id, instantiated in this case to stage m1.2. This filter checks its stage parameter is either enacted or able to be restarted (not yet completed) and, if so, passes received event flows into action “Notify Role”. This action, parameterised by a role, informs the person or people filling that role (in this example, the “coders” of m1.2), that testing has started or finished. Notification is, by default, via an e-mail like message, but users can specify, via dialogue box, they wish to be informed in other ways, such as by opening a dialog on the coders' screen, shading the “m1.2:implement changes” stage icon to indicate presence of a message, etc (Grundy et al., 1996a). The righthand event flow from m1.3 notifies coders if testing has finished in either the “fix code”, “fix design” or “finished testing” states. The coders are again only notified if the m1.2 stage has not completed.

“Notify Role”, “Made Current” and “Not Complete” are reusable (system defined) filters/actions while “finished testing” is specific to this particular process model. Filters are defined by a pattern-matching language (for simple filters), instantiation of a reusable template (via a forms interface), or use of an API to Sernipity's implementation language (see later in this section).

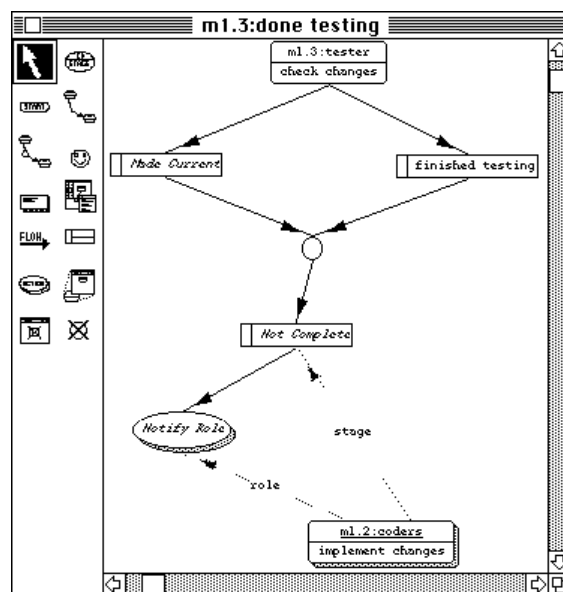


Figure 8. An example of process stage event filtering.

4.2. HANDLING ARTEFACT, HIERARCHICAL AND MULTIPLE EVENTS

To allow different processing based on the kinds **or** numbers of events, VEPL allows users to: distinguish between enactment and tool/artefact update events; specify if subprocess component events are handled; and specify that a filter/action is to handle sequences of, rather than individual, events. Figure 9 shows these additional modelling capabilities.

Notational Symbol	Example	Description
		The filter or action is informed of any artefact update (or tool/role) events, rather than enactment events. In the example, FilterName is informed whenever an artefact update event occurs and process stage 1.1 is the current enacted stage (i.e. the stage on which the user is currently working). If the \mathcal{A} was not present, the filter is only be informed whenever stage 1.1. is enacted or de-enacted.
		The filter or action is informed when stage 1.1. or any of its subprocess stages are enacted or de-enacted. In the example, FilterName is informed whenever an enactment event occurs for stage 1.1 or any of its subprocess stages. If \mathcal{A} and Σ are combined, the filter is informed of any artefact update events made when stage 1.1. or any of its subprocess stages are the current enacted stage.
		The filter or action is informed of a series of events from the connected process stage (or artefact, tool or role), as opposed to being informed of single events. In the example, several enactment or deenactment events may occur, all of which are sent to FilterName and it will decide on the action it will take based on this sequence of events, not just as each event is received. The * is often combined with Σ and \mathcal{A} annotations.

Figure 9. Extended modelling capabilities of VEPL.

As an example, figure 10 specifies that the “designer” associated with “m1.1:design changes” is to be notified of any schema updates made by “coders” associated with “m1.2:implement changes”. Artefact updates (indicated by the \mathcal{A}) made by any user filling the “coder” role while working on m1.2, or any of its subprocesses (indicated by the Σ), flow into a filter (“RDB table change”), which checks if the change is to a table. If so, the “designer” is notified of the event. “RDB table change” can be defined in two ways: as a large list of artefact change event patterns which indicate the change is to a table, or by determining if the event originated from the RDB table designer tool.

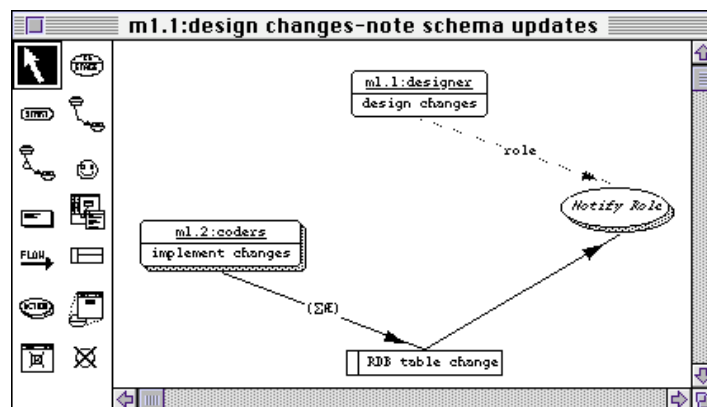


Figure 10. An example of artefact change event filtering.

Roles, tools and artefacts used in a VEPL specification act as filters, if events flow into them from other process model components. For example, Figure 11 specifies that “rick” is interested in any changes “john” makes to the “customer” table, and “rick” is to be informed asynchronously of these changes by storing them in a change list. Changes from m1.2 flow to the “customer table” artefact, which here acts as a filter, only passing on change events for this artefact. Filter “Made By” checks the changes were made

by user “john”, and action “Store Event” records the event in a change list, named "john's changes" owned by “rick”. Artefact update events can also be filtered through roles and tools. The m1.2 stage can be removed from the example in figure 11, and the flow from artefact “customer table” annotated with \mathcal{A} . This would then specify that any change at all to “customer table”, made by “john”, should be stored for “rick”, regardless of the plan stage it was made in.

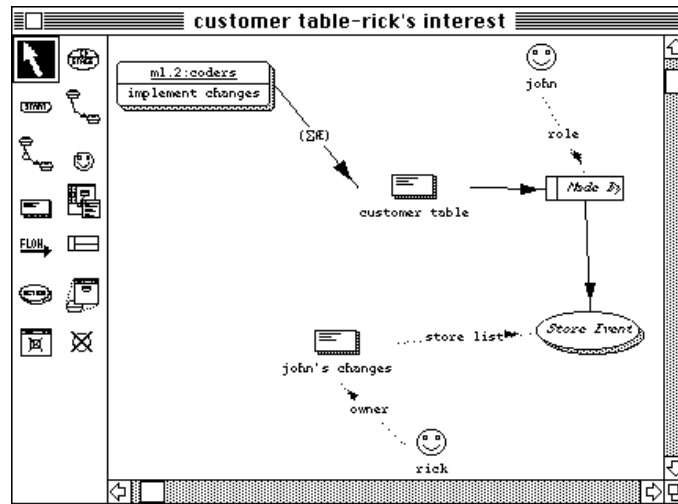


Figure 11. An example of complex filtering and actions.

Event flows annotated with a * indicate multiple events from a stage, artefact, tool or role are to be processed. The attached filter/actions receive multiple events before deactivation, allowing them to recognise and process complex event sequences, and maintain state between the individual events. They can thus act as semi-autonomous agents. Such filter/actions reset their state after receiving the sequence of events they are interested in. For example, in Figure 12 the “Add Entity & Rel/Roles” filter recognises a sequence of ER updates (addition of a new entity, relationship, roles and attributes) and notifies coders of the entire table update using a single abstract message, rather than via a long list of messages describing each discrete change made. In this case, the filter accepts ER modeller artefact update events until it recognises an unrelated artefact (e.g. a different table) is being modified. It then passes on a single event to “Notify Schema Affected” and resets its state.

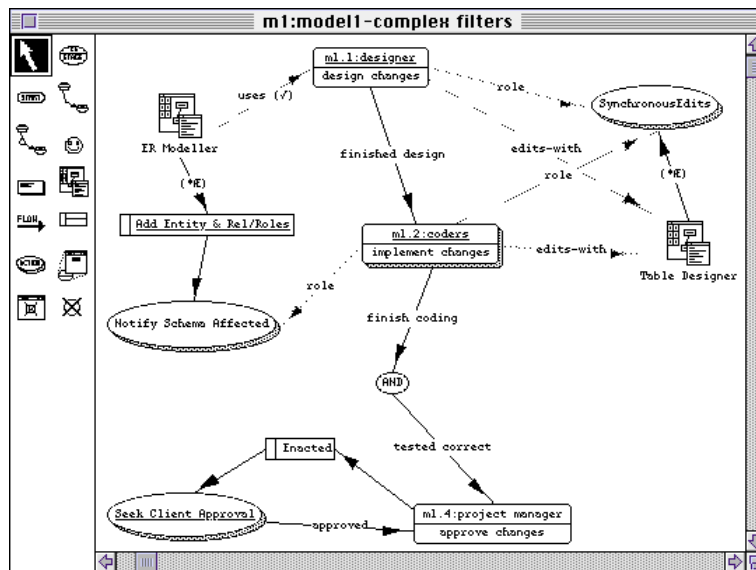


Figure 12. Multiple event processing, stage coordination and external process interfacing.

Another example of a complex filter/action is the “SynchronousEdits” action, used in figure 12 to specify how work is to be coordinated between people filling the roles for two concurrently enacted stages. In this case, if the m1.1 and m1.2 stages are concurrently enacted and the roles involved both use the “RDB Table Designer” tool, they must use the synchronous editing mode of that tool to ensure schema changes they are making do not conflict. The “Seek Client Approval” action models a non-computerised process, asking the “project manager” to liaise with the customer and gain approval for the new system changes made.

4.3. DEFINING FILTERS AND ACTIONS

Template Filter/Action	Kind	Parameters	Description
Enacted	filter		On receipt of an event, checks if this event is a “start” (i.e. stage enactment) event.
Deenacted	filter		On receipt of an event, checks if this event is a “finish” (i.e. stage deenactment) event.
Made Current	filter		On receipt of an event, checks if this event is a “made_current” (i.e. stage made the current enacted stage) event.
Finished Current	filter		On receipt of an event, checks if this event is a “finished_current” (i.e. stage finished being the current enacted stage) event.
Not Complete	filter	stage	On receipt of an event, passes on event if the given stage has not been completed (i.e. stage’s “completed” flag not set).
Complete	filter	stage	On receipt of an event, passes on event if the given stage has been completed (i.e. stage’s “completed” flag is set).
Made By	filter	role	On receipt of an event, checks the event was caused by the specified role(s).
Edited By	filter	tool	On receipt of an artefact update event, checks the event was caused by editing using the specified tool(s).
Notify Role	action	role	On receipt of an event, notifies the people filling the role of the connected process stage(s) that the event has occurred.
Store Event	action	update list	Stores the event received in an “updates list” artefact (a list of event record descriptions).
Wait	action		On receipt of event, queues the event and does not forward it to connected stages, filters or actions until the user-specified period to wait has completed. Multiple events can be queued using the * annotation on the event flow.
LaunchApp	action	tool	Launches the specified tool.
CreateDoc	action	tool, artefact	Instructs the specified tool to create a new artefact of the specified type & name.
OpenDoc	action	tool, artefact	Launches the specified tool (if not already running) and instructs it to open the specified artefact(s).
SaveDoc	action	tool, artefact	Instructs the specified tool to save the specified artefact(s).
CloseDoc	action	tool, artefact	Instructs the specified tool to close (i.e. stop working with) the specified artefact.
QuitApp	action	tool	Instructs the specified tool to quit.

Figure 13. Some of the template filters and actions provided by Serendipity.

Some of the filters and actions shown in the preceding examples are reusable library templates, some are user-defined by simple specification of patterns to match, some are defined by subprocess models, and others utilise an API interface to the implementation language of Serendipity. Examples of commonly-used template filters and actions are shown in Figure 13. Users can extend the library by adding their own filters and actions.

VEPL filter/actions can be defined using VEPL itself by expanding a subprocess, parameterised by start and finish stages, similar to EVPL stage subprocesses. The filter or action icon is then connected to appropriate process model components, and when it receives an event its subprocess is used to determine the response to the event. Figure 14 is a generic filter/action “update data model”, which checks if a change to a given “entity” artefact is an entity rename or attribute modification (add, delete, rename or change type), and, if so, stores the artefact change event in an “updates list” specified. The finish stage “update” represents an output from this filter/action, allowing the event to be flowed onto other filters/actions. This filter/action is reused by adding an action icon “update data model” to a process model view. The user then specifies an event flow from an entity artefact into “update data model”, and a

usage flow into “update data model” from an “updates list” artefact. The event flow out of “update data model” can optionally be used, as required.

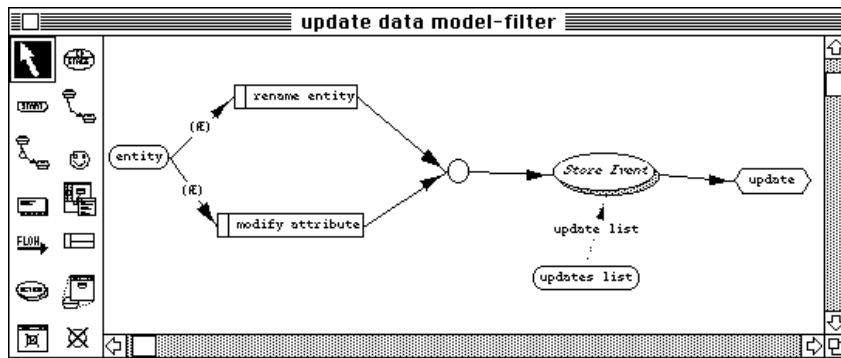


Figure 14. A parameterised filter/action.

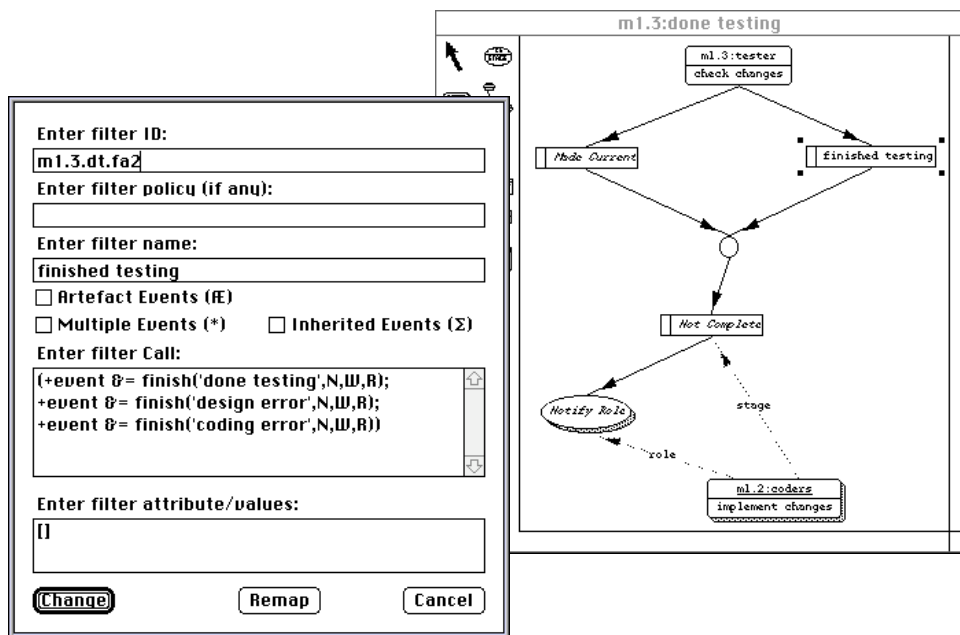


Figure 15. Example of specifying detailed filter and action information.

Filters and actions can also be implemented via a Prolog-based API to Serendipity's implementation. Pattern-matching on input event representations is used to differentiate between different event types. Using the API, filters or actions may call internal functions or access the internal data structures of Serendipity to implement complex filtering operations, or actions.

Figure 15 shows the specification of filter “finished testing”. Names and ids are specified in the top three fields. Flags are set to specify the types of event the filter is interested in. The filter call field is either blank (if the filter action is copied from a template or has a subprocess model), or contains Prolog calls. The “&=” operator used in “finished testing” compares an enactment event received by the filter (indicated by “+event”) with the given pattern. In this example, the filter responds to finish events in the states 'done testing' 'design error', or 'coding error'. Various other operators and values can be used when specifying the filter/action API call(s), such as the \$= operator which compares artefact events to a pattern, the +self value (the object ID of the filter/action receiving the event), and the +from value (the object ID of the model component the event is from). Any parameter inputs to the filter/action specified by usage connections are referred to by +UsageFlowName, where UsageFlowName is specified in a dialog for the usage connection when it is added.

The bottom edit box (“Enter filter attribute/values:”) is used to specify state information for a filter/action. This is usually used to constrain or modify filter/action behaviour. For example, for the “Notify Role” action, this has the value “(notify_method=message)”, specifying that for this instantiation of the Notify Role template action, coders should be notified by a message (using an email-like messaging system described in Section 5). Other values of this notify_method attribute include: “open_dialog”, to notify users via a dialogue; “highlight”, via highlighting a process stage icon; or “broadcast”, via a broadcast message appearing at the bottom of the other users’ screens.

5. The Serendipity Environment

We have developed an environment for Serendipity supporting EVPL and VEPL. This provides multiple views of process models, allows processes to be enacted, supports process improvement and reuse, and allows meta-process models to be defined to describe and control the process modelling task itself.

5.1. MULTIPLE PROCESS MODEL VIEWS

EVPL and VEPL process model descriptions are produced using the same editing tool. Several views of the same process model can be defined, each adding more information about the model as a whole. For example, Figure 3 is a process model for software system enhancement, Figure 6 describes the artefacts, tools and roles used this process model, and the figures in the previous section associate filters/actions with these stages, artefacts, tools and roles. Textual views of process stages are also available (Figure 6). Serendipity keeps all views of a process model consistent under change, or at least, in a known state of inconsistency (Grundy et al., 1995a).

5.2. PROCESS ENACTMENT

A Serendipity process model can be enacted by users for a particular project at any time. Enacted processes can be modified at any time to extend and improve the process specification, more precisely define the work to be done, or rewrite the history of work done to assist in process improvement and work documentation.

A process model is enacted by selecting the stage to enact and selecting a menu option. The user also specifies a reason for the enactment and, if the stage has subprocess models defined, a start stage to commence in. An enactment event is then sent to the process stage, and start stage (if specified). The start stage immediately sends enactment events to all stages connected to it. As work on a stage is completed or a subprocess completes (a finish event flows into a finish stage), finish events flow into connected stage(s), enacting them. AND stages wait for all stages which flow into them to complete; OR stages enact all stages they flow into upon receipt of a finished event from any input stages. Users can manually complete enacted stages (if no actions prevent this) by supplying a finish state for them.

Figure 16 shows an enacted process model. Enacted stages (ie those started and not yet finished) are shaded. Event flow(s) which have activated a stage are also highlighted. The user's current enacted stage (the stage that user is currently working on) is bold highlighted. Users can select any enacted stage they fill the role of to be their current enacted stage. Users can view a list of their enacted stages, also shown in the figure, and this can function as a “to-do” list for that user.

When a stage is enacted, finished, or made/unmade current, it records this event in an enactment history. A reason is also stored specifying when and why the event occurred and which user or stage caused the event. A modification history describing changes to each stage and view is also stored, allowing users to review the evolution of process models. Figure 16 illustrates these two histories for stage m1.1.

“EXCEPTION” actions can be defined for process models. These have no input event flows, but are automatically invoked when a stage, action or filter finishes in a state not handled by the process model i.e. no output event flows match the finishing state name. This usually indicates an error in the process model. “EXCEPTION” actions reduce the complexity of process models by eliminating many error-handling actions and event flows. Multiple “EXCEPTION” actions can be defined for a process model with filters used to determine which one handles a particular exception. If not defined for a model a default handler informs the user of a process model exception (via a change description in a dialog). The user can then handle the exception manually, or extend the model to take the exception into account.

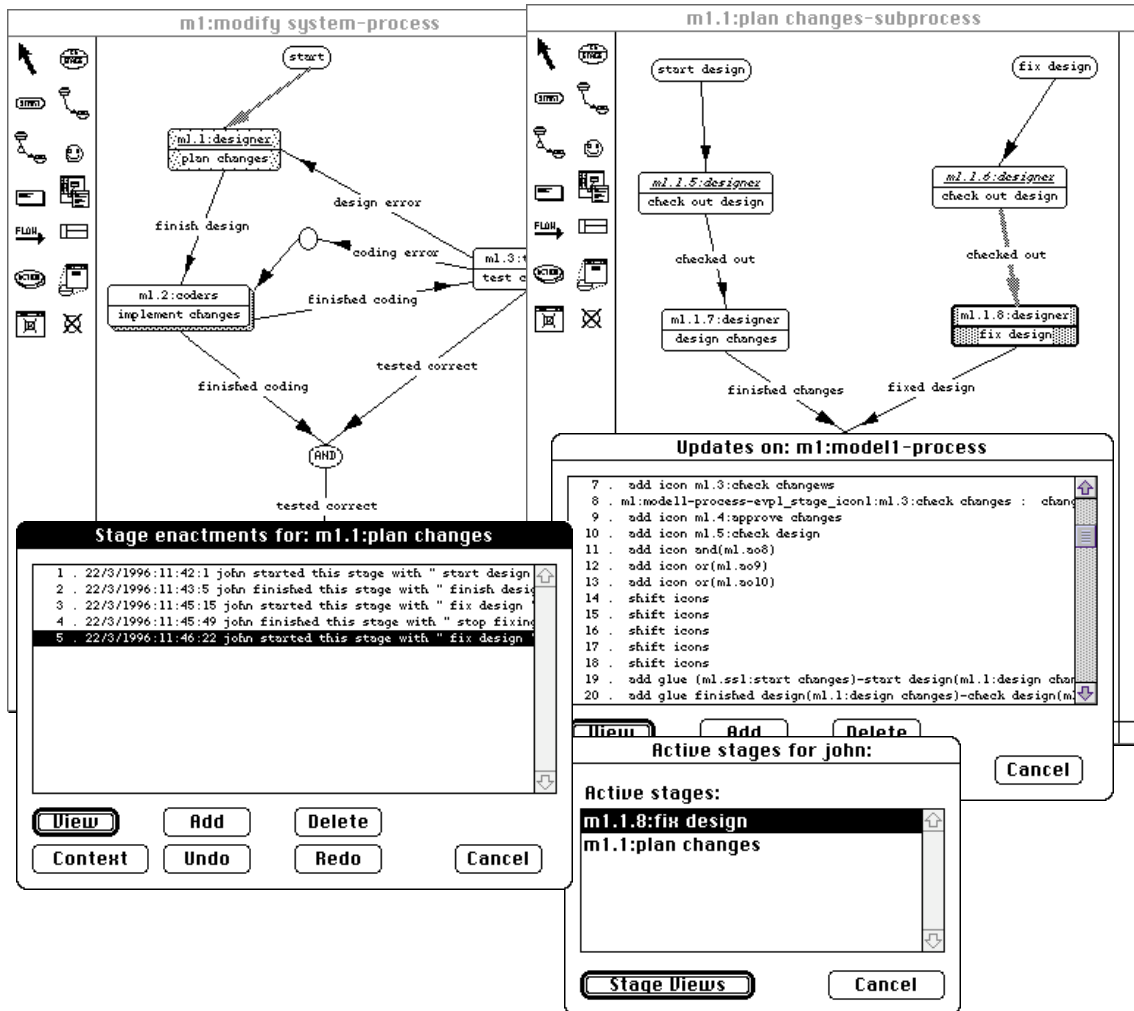


Figure 16. Enactment and modification histories for a stage, and a user’s to-do list.

5.3. COLLABORATIVE PROCESS MODELLING AND WORK PLANNING

Serendipity process model views may be shared amongst developers and synchronously, semi-synchronously or asynchronously edited. Serendipity has also been integrated with several small CSCW tools which provide work context-dependent notes, messaging and text chats to facilitate inter-person communication (Grundy et al., 1996b).

To collaborate effectively on a large project, users need higher-level awareness support in addition to the low-level messaging, communication and view editing mechanisms (Grundy et al., 1996b). Serendipity provides information about the work contexts’ of their collaborators in a high-level, synchronous way as shown in Figure 17, by highlighting and colouring collaborators enacted stages. Actions can also be specified to highlight artefacts currently being modified by other users, such as the “changes list” in

Figure 17. Users may have more than one EVPL view open, with enacted and current enacted stages highlighted.

5.4. PROCESS IMPROVEMENT, REUSE AND META-PROCESS MODELS

Process model often need modification during or after use, as exceptions or unforeseen events arise, and may evolve as users become more proficient at describing their work processes, or because details of a process used varies between projects. Serendipity allows process models to be modified at any time: before, during or after use.

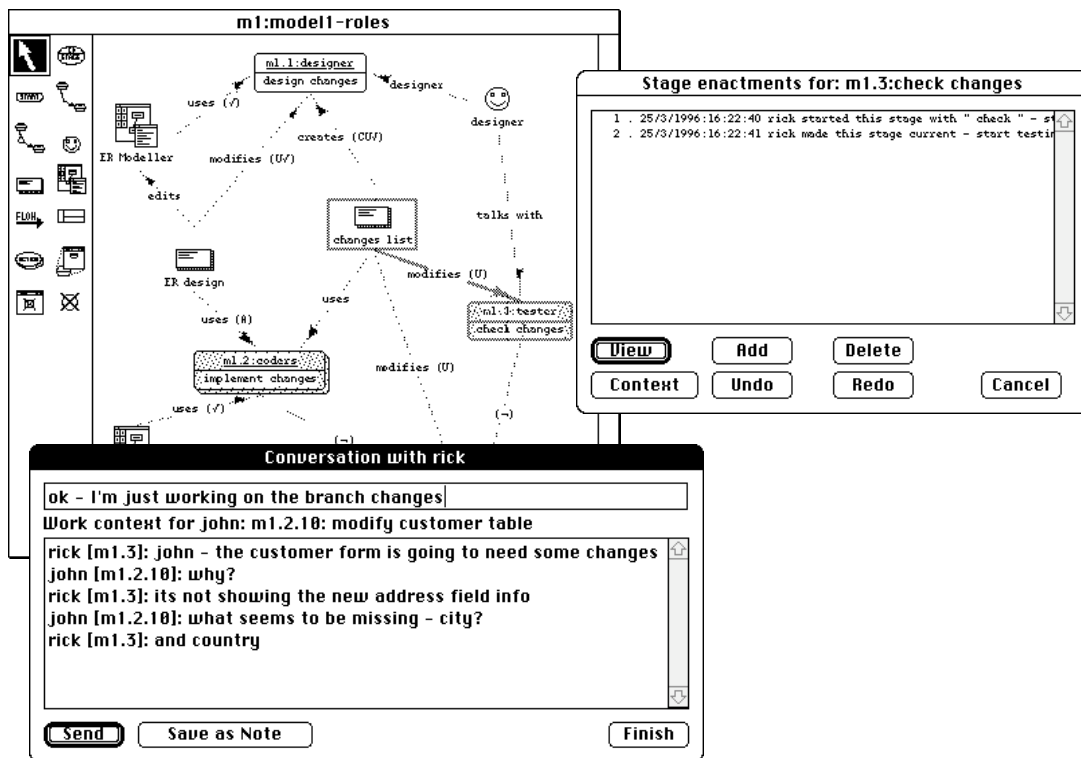


Figure 17. An example of highlighting another user’s current process stage and artefacts.

To aid in model evolution, Serendipity can provide statistics about the complexity (number of substages, event/usage flows, etc.) of subprocess models, the number of enactments a stage or subprocess model has had, the amount of time each stage has been the current enacted stage, the number of modifications made to models, and the number of work artefact changes made while a stage or its subprocesses were the current enacted stage (Grundy et al., 1996b).

Process stages or filter/actions can be abstracted into process templates. Users can view and edit template models but templates cannot, however, be enacted. Templates are abstract and must first be instantiated (copied) by adding a stage to a process model, specifying it is based on a template.

To guide the process modelling, enactment and improvement process itself, meta-process models can be defined as EVPL and VEPL models. These are also useful for coordinating the modelling activity between multiple collaborators. An added advantage is that changes made to process models can be recorded against the current enacted stage for the meta-process model, allowing collaborators to track reasons for process model changes. Meta-process models also allow users to specify event handling and rules for the EVPL and VEPL notations themselves, by specifying filters and actions on artefact events for process models (i.e. handling events from process model artefacts and tools). Examples of such event

handling/rules include specifying ownership and access privileges for process models, controlling when and how process models can be updated, and specifying default exception handling approaches.

6. Using Serendipity with Other Tools

Serendipity provides more than a process modelling and management environment. Used with other tools it specifies the *context* of work for multiple collaborators, facilitating activities such as collaborative software development, method engineering and office automation.

A user's current enacted stage defines the work context for that user. Any work artefact changes made using other tools are assumed by Serendipity to be part of the work associated with this stage. Changing the current enacted stage changes the context of work. To illustrate the benefits of this, we briefly describe how Serendipity provides a process modelling tool for an integrated Information Systems Engineering Environment (ISEE) (Grundy et al., 1996c). This environment includes the SPE object-oriented software development environment (Grundy et al., 1995b), an ER modelling tool (MViewsER) (Grundy and Venable, 1995a), a form/report building tool (MViewsDP) (Grundy et al., 1996d), and a NIAM modelling tool (MViewsNIAM) (Venable and Grundy, 1995).

Serendipity does more than just define the work context for a user. As shown in Section 5, stages record their enactment and modification histories. They also record the work artefact updates made while the stage is the current enacted stage, thus documenting the history of work for this work context. Figure 18 shows an example of changes made to an MViewsER view stored for the “m1.2.10:modify customer table” stage for coder “john”.

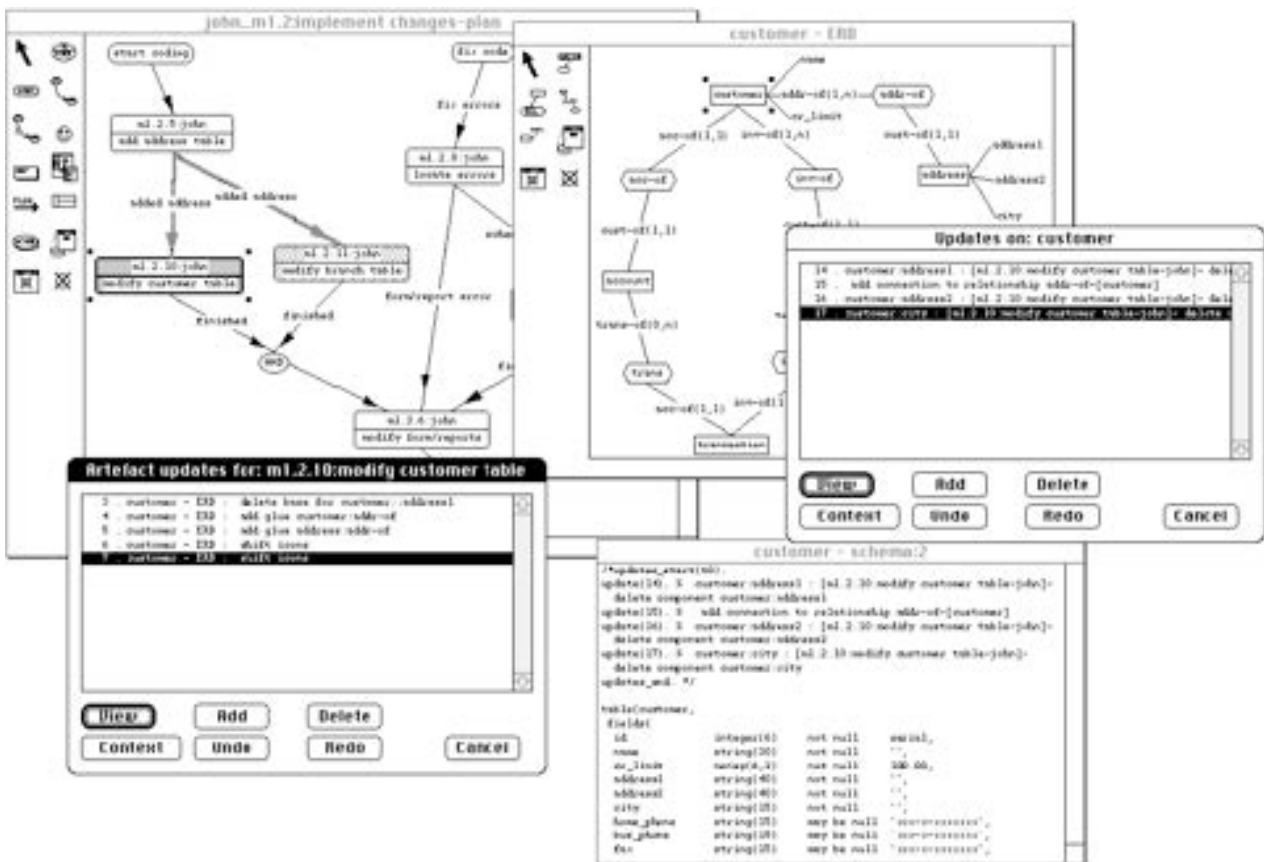


Figure 18. A work artefact change history in Serendipity and work artefact modification history in MViewsER.

In addition to recording work artefact updates in Serendipity stages, work artefact change descriptions stored by the tools themselves are augmented with information about the work context they were made in. Users of these tools can review work artefact modification histories to determine the changes made, who made them, and when and why they were made (i.e. the work context they were made in). Figure 18 shows an example of a modification history for the “customer” entity/schema from the MViewsER modelling tool. The augmented change descriptions for “customer” are also shown in its schema view, “customer - schema”. Note that the changes shown in the “Updates on: customer” dialog and “customer - schema” view are customer artefact updates. Those shown in the “Artefact updates for: m1.2.10:modify customer table” are changes made to the “customer - ERD” view. The user can access the artefact-level updates made in response to these view changes via the “View” button.

While Serendipity highlights process stages, usage connections and artefact and tool icons in use by another developer, as shown in Figure 17, it also allows actions to be specified which highlight work artefacts in integrated tool views (Grundy et al., 1996b). Filter/actions defined by a developer can detect work artefact changes made in an integrated tool by someone else, and send messages to the tool of the developer to highlight these icons. Due to this degree of integration of Serendipity and our ISEE, users can also undo or redo work artefact changes within Serendipity or the tools which generated them. Users can also select a work artefact change and request a list of all views of the artefact in the available tool(s). In a tool, users can request a list of views of the appropriate work context of a selected change description.

Serendipity actions can be used to constrain some of the integrated tools' functionality, facilitating “Method Engineering” within our integrated environment. Most software development notations and methodologies are designed to be generic across all problem domains. Research indicates, however, that configuring notations and methodologies to the needs of the particular project results in more appropriate tools and techniques (Harmsen and Brinkkemper, 1995). Computer-aided Method Engineering (CAME) tools support this by allowing system developers to specify which notations (or parts of notations) and methodologies (or parts of methodologies) are to be used for a particular project (Harmsen and Brinkkemper, 1995). We have used Serendipity to support Method Engineering for our ISEE (Grundy et al., 1996c). Serendipity EVPL views specify tool usage and process models (“methodology steps”). VEPL views specify rules and event handling which restrict the use of certain tools and guide or enforce the methodology processes.

Serendipity has also been used to support process modelling for a suite of office automation programs, including Macintosh versions of Microsoft Word™, Microsoft Excel™, the GlobalFax™ fax/OCR application, and the Eudora™ email utility.

7. Architecture and Implementation

Serendipity and the tools that we have integrated to make up the ISEE described in Section 6 are implemented using the MViews architecture for developing ISDEs. Integration of Serendipity and these tools, although all independently developed, has proved very successful, due to the component and event-based nature of MViews. The integration between Serendipity and the Office Automation tools was more limited, due to the more limited interface provided by these tools.

7.1. MVEIEWS

MViews is a framework of object-oriented classes which provides abstractions for implementing ISDEs (Grundy and Hosking, 1996; Grundy and Hosking, 1993; Grundy et al., 1996d). New environments are constructed by specialising classes to describe the ISDE repository and view representations. Software system data is described by a graph-based structure, with graph *components* (nodes) specifying e.g. classes, entities, attributes and methods, and *relationships* (edges) linking these components to form the

system structure. Multiple views of this repository are defined using the same graph-based structure. These views are rendered and manipulated in concrete textual and graphical forms. External tools not built using MViews can be interfaced to the integrated environment by using “external view” data and event translators. Figure 19 shows an example of the MViewsER Entity-Relationship and relational schema modelling tool developed using MViews (Grundy et al., 1995b). The repository describes entities, relationships and attributes, entity-relationship connections, and schema text. The multiple views provided by MViewsER include graphical ER views and textual schema and documentation views.

MViews uses an event-based software architecture. This supports inter-component consistency management by generating, propagating and responding to *change descriptions* whenever a component is modified. A change description documents the exact change in the state of a component. It is propagated to all relationships the component participates in. Receiving relationships can respond to a change description by applying operations to themselves or other components, forwarding the change description to related components, or ignoring the change. This technique supports a wide variety of consistency management facilities used by ISDE environments, including multiple view consistency, inter-component constraints, efficient incremental attribute recalculation, undo/redo, and version control and collaborative facilities (Grundy et al., 1996d).

MViews is implemented in Snart, an object-oriented extension to Prolog. Environment implementers specialise Snart classes to define new environment data dictionaries, multiple views, and view renderings and editors (Grundy et al., 1996d). Snart is a persistent language, with repository and view objects dynamically saved and loaded to a persistent object store.

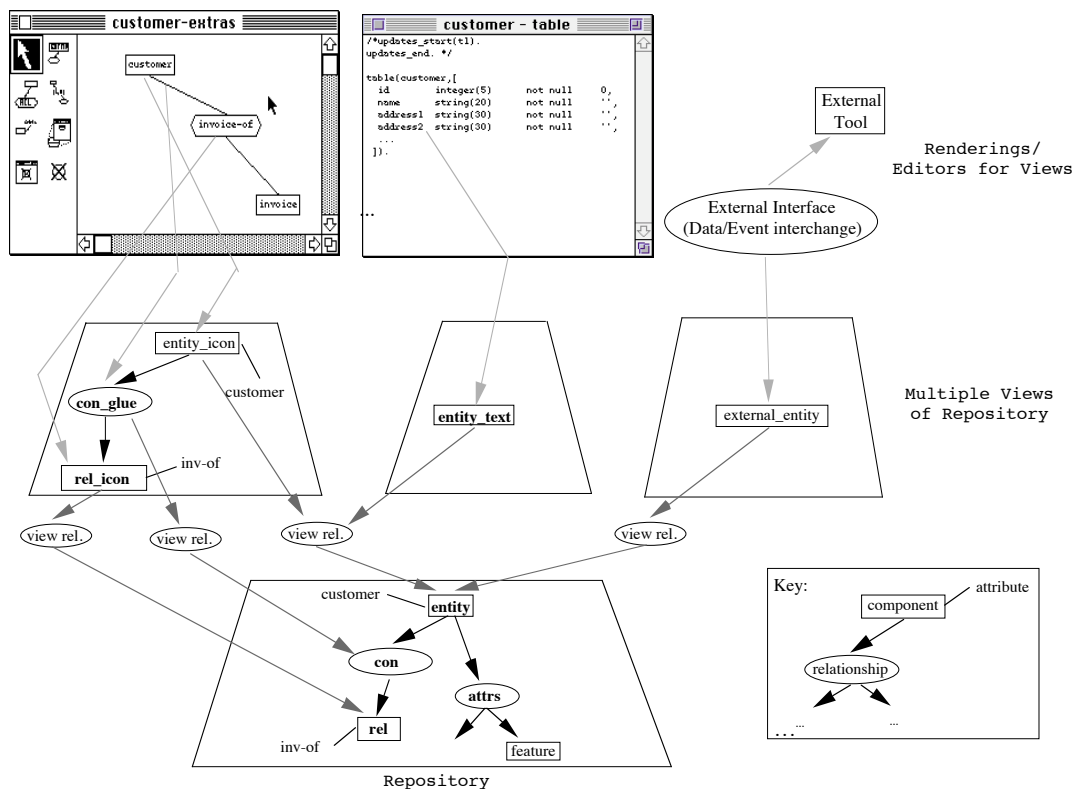


Figure 19. Example of implementing MViewsER using MViews.

7.2. INDIVIDUAL MODELLING TOOLS

We have built many ISDEs using MViews including SPE (Snart Programming Environment), MViewsER, MViewsNIAM and MViewsDP, the tools integrated into the ISEE. Integration of these tools was achieved using integrated, hierarchical repositories (Grundy and Venable, 1995a; Grundy and

Venable, 1995b). This involves either defining integrated repositories which represent, for example, OOA, NIAM and ER data in a common representation, or linking individual repository items with MViews inter-repository relationships, such as between MViewsER schema fields and MViewsDP form components. Inter-repository links keep data in different repositories consistent under change in a similar manner to the view relationships described above.

7.3. THE SERENDIPITY ENVIRONMENT

Serendipity was built by specialising MViews classes for representing repository components and relationships, view icons, glue and textual components, and view editors. Constructing EVPL and VEPL diagrams using Serendipity view editors results in construction of repository-level components and relationships describing a process model.

When a process model is used, process model component state variables are modified to indicate receipt of enactment events. Enactment events are simply represented as MViews change descriptions. These are propagated along event connections, with stages interpreting event change descriptions they receive (AND, OR, and start/stop stages interpret these in a special way). Filters and actions interpret change descriptions by comparing the event change descriptions to a filter pattern, running a Prolog predicate which accesses MViews data (i.e. the API interface), or enacting a subprocess model.

Process stage enactment and modification histories are provided by MViews. Artefact modification histories are constructed as stages receive artefact change descriptions from other MViews environments, or from components which interface to applications not built using MViews and which translate external tool events into MViews change descriptions.

7.4. INTEGRATING SERENDIPITY WITH OTHER ENVIRONMENTS

Figure 20 shows the way Serendipity interfaces with other MViews ISDEs and external tools. If a change is made to a view-level ISDE tool item (1), this is propagated to a repository (“base view”) level item change, and the change description generated by this repository-level item change propagated to the ISDE base view component (2). The base view adds artefact, tool and user information to the change description, and broadcasting of this modified change description is detected by the Serendipity environment base view (3), which then forwards the change description to the user's current enacted stage (4). This, in turn stores the change in its artefact modification history (5) and forwards it to any filters/actions it is linked to by artefact update event connections (6). The process stage attaches “work context” information to the artefact change description (primarily information about the stage itself), sends it back to the Serendipity base view (7), which returns it to the ISDE tool base view (8). The ISDE base view returns the augmented change description to the work artefact which generated it (9), which stores it in its own modification history (10). View updates are sent to Serendipity in the same manner, with the updated view component sending a change description to the view (11), which forwards it to its base view (12), with the base view forwarding the change description to Serendipity and receiving an augmented change description as before. Any change to the enactment status of Serendipity base process model components (17) is detected by their view components (18), which are rerendered to reflect the change.

External tools, such as Microsoft Word™ or Excel™, can also use Serendipity to provide a process modelling and work context environment. Events from such external tools (Apple Events) are sent to a small translator program (13), which forwards them to Serendipity for processing (14). Serendipity events can be sent to the external tools via the translator (15, 16). These might request the external tool be launched, ask it to save, open or close files, or ask it to carry out some other task (such as send a fax or an email message).

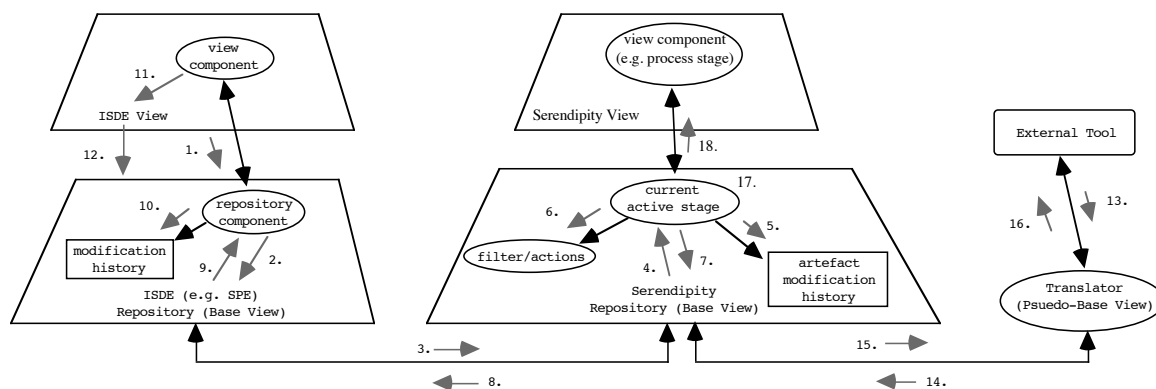


Figure 20. Integrating Serendipity with integrated tool views.

Serendipity's collaborative editing facilities are provided by the C-MViews extensions to MViews (Grundy et al., 1995c). Filters and actions specified on process models work as expected if synchronous editing is used as all collaborators share the same process model data. Asynchronous editing implies collaborators have different process model versions which they edit and enact independently, with other collaborators not being aware of these events. Semi-synchronous editing propagates editing, enactment and artefact update events between the environments of collaborators who express interest in these changes for particular process stages, tools, artefacts or roles. C-MViews base views have been extended to incorporate a notion of *monitors* which detect events of interest to collaborating users. Users request, via the C-MViews server, other collaborators' environments to establish monitors on items of interest. Events are then propagated semi-synchronously, via the server, to the interested user's environment for presentation and/or actioning.

The CSCW messaging and note annotation tools provided by Serendipity are separately-developed tools which can be used with any MViews environment. The context-sensitive messages store information about the MViews component(s) they are related to, and notes are connected to specified components by relationships. The high-level highlighting of Serendipity process model stages enacted by other users is implemented by having the central C-MViews server broadcast all enactment events to each user's environment. This then records which stages other users have enacted or deenacted, and highlights these on visible process model views (if requested by the user). We slightly modified these tools to utilise information about the current enacted Serendipity process stage for users when storing notes, sending messages or sending text chat lines, thus supplying work context information for inter-person communication and notes.

8. Discussion

Our work has made four main contributions to Process Technology. EVPL provides an expressive, graphical process modelling and work planning language, which can be animated to support both high-level and low-level work context awareness during cooperative work. VEPL is a novel, graphical event handling language that is both accessible for simple use by inexperienced process modellers, and yet expressive enough for experienced process modellers and environment implementers to build sophisticated event handling systems. The Serendipity environment provides multiple view support for building, enacting, reusing and improving EVPL and VEPL process models, and includes a range of cooperative work capabilities. Our integration of Serendipity with other tools, particularly software development and CSCW tools, provides novel user interface capabilities, with Serendipity allowing EVPL and VEPL to utilise events from these integrated tools in various ways.

8.1. EXPERIENCE WITH SERENDIPITY

Serendipity has been applied to a range of small to medium process modelling, work planning and coordination, and tool integration problems. The authors have built a number of process models describing: software processes; coordination of collaborative software development; coordination of use of disparate office automation applications; general work processes for academics, students and business people; and for meta-models for method engineering and process improvement. Serendipity has also been used by colleagues and students to describe: software processes; steps in different Information Systems methodologies; and general work processes. The largest process model so far developed has over 200 process stages, artefacts, tools and roles, with over 60 process model and filter/action views. A project is currently in progress to more formally compare Serendipity's languages to other workflow and process modelling languages and to evaluate the Serendipity environment facilities against those of other workflow management systems and process-centred environments.

Serendipity environment performance over the above range of tasks has generally been good, but with some deficiencies, elaborated below. The asynchronous and semi-synchronous CSCW tools and editing capabilities are quite usable in our current implementation, but the synchronous tools have prohibitive performance problems (Grundy et al., 1995c). Processing stages with several filters and actions also cause performance problems, particularly with hierarchical filters (Σ annotation on event flows), which must receive events from all subprocess components. Users can request that filters and actions be actioned semi-synchronously and that the environment process filters for specified stages in idle time. These techniques improve performance with the drawback that enforcement strategies are applied some time after enactments or artefact changes have been made, necessitating occasional roll-back of artefact changes.

Integration of Serendipity with other, independently developed, MViews tools has been successful, producing tightly-integrated environments. Serendipity actions can send events to other MViews tools to perform almost any operation, and any tool event can be detected and acted upon by Serendipity filters. This is due to the component and event-based architecture of MViews systems, and the common implementation language and view and repository representation techniques. Integration with heterogeneous tools not implemented with MViews has resulted in more limited forms of integration. Serendipity actions can be defined to communicate with these tools but for significant levels of integration, this requires extensive coding and still results in less than complete integration. Serendipity currently runs on the Macintosh only. While we have integrated it in a limited way with a number of commonly-available Macintosh applications, it is much more difficult to communicate with applications on other platforms.

8.2. COMPARISON TO OTHER LANGUAGES AND SYSTEMS

Process modelling languages need to support the description of stages in the process, the data used by process model stages, and the handling of events related to stages, work artefacts and tools. Many process modelling languages use only textual languages to codify tasks and their data. Examples include those of Marvel (Heineman et al., 1992), Adele (Belkhatir et al., 1994), ConversationBuilder (Kaplan et al., 1992a; Kaplan et al., 1992b), Oz (Ben-Shaul and Kaiser, 1994a) and EPOS (Jaccheri and Conradi, 1993; Conradi et al., 1994). Work artefacts are usually coded as types, for example in EPOS these are "dataentity" specialisations and in Adele they are aggregate type relations. Steps in a process model are usually encoded as rules, for example in EPOS as task types and Adele as event triggers. While these languages allow users to precisely specify process model data and activities, they are difficult for many users to understand and modify (Baldi et al., 1994; Bogia and Kaplan, 1995; Swenson et al., 1994), and cannot readily be utilised for high-level work context awareness, such as via animation. Our EVPL and VEPL process modelling languages have the same expressive power as these textual languages, but use a primarily graphical approach to visualising process model activities and data (with detailed process stage,

artefact, tool and role characteristics specified in forms and textual views). EVPL diagrams provide significantly clearer representations of process model structure than their textual counterparts, as process modellers can see multiple levels of abstraction using EVPL diagrams. VEPL provides a graphical approach to expressing event handling, allowing process modellers to better visualise event flow and handling than a comparative textual encoding.

A variety of graphical process modelling languages have been developed for use by process-centred environments and workflow management systems. Examples include E³ p-draw's E³ PML (Baldi et al., 1994), SPADE's SLANG (Bandinelli et al., 1994; Bandinelli et al., 1993; Bandinelli et al., 1996), ProcessWEAVER's transition nets (Fernström, 1993), Action Workflow's loops (Medina-Mora et al., 1992), Regatta and TeamFLOW's Visual Planning Language (VPL) (Swenson, 1993), and wOrlds's obligation nets (Bogia and Kaplan, 1995). Graphical modelling capabilities of some of these systems, such as VPL, SLANG and wOrlds, is limited to specification of the stages in a process and the "enactment flow" between these stages. For example, SLANG encodes activity structure as a form of petri-net, but textually encodes activity artefacts and tools as process types and transitions as guards and actions; VPL represents process stages graphically with interconnecting flows, but lacks representations of stage data and enactment rules; and wOrlds represents aspects of social structure graphically, including obligations between "process stages", but provides only a simplistic graphical representation of process data and obligation rules. Our work, and work with E³ PML, ProcessWEAVER and Action Workflow, has found graphical modelling of process stage data (artefacts, tools and roles), and the visualisation data shared by multiple process stages, to be very important when using complex process models. E³ PML uses an object-oriented process modelling notation, with a variety of representations (stages, artefacts, tools), and interconnections (task decomposition, inheritance, control flow). E³ PML represents task decomposition on the same diagram, whereas EVPL utilises multiple overlapping and hierarchical views, making decomposition easier to manage and resulting in clearer process models. E³ PML also lacks the range of usage connections and annotations of EVPL, making it less expressive. ProcessWEAVER (Fernström, 1993) uses a transition net (a form of petri-net) to model cooperative procedures via tasks and subtasks, with token flow driving enacted process advancement. The transition nets do not explicitly represent artefacts, tools and roles, but nodes can be defined which correspond to the manipulation of them. ProcessWEAVER nets thus do not make clear the roles involved in stages of a process, nor the artefacts and tools, nor do they indicate usage of this work context information, as in EVPL. Action Workflow (Medina-Mora et al., 1992) uses a document flow model to represent work processes, which has been found to be deficient for many process modelling situations (Swenson et al., 1994). EVPL uses a more flexible and expressive state-transition model, as do VPL, ProcessWEAVER and SPADE, with EVPL processes driven by the propagation of enactment events. This notion of explicit enactment events in EVPL, rather than state transition by token propagation in SPADE and ProcessWEAVER or the use of speech acts in VPL, is important, as it is utilised by VEPL to handle enactment, artefact update and tool events in a homogeneous fashion.

The event-handling of most graphical process modelling languages is codified graphically for "enactment" events (state transitions), but textually for stage guards and actions, and for coding interaction with people or other tools. Most workflow management systems, such as Action Workflow, Regatta and TeamFLOW, provide a limited range of "interesting events", supporting interaction with other tools and some forms of notification and work coordination based on event occurrence. These are usually codified using a form-based approach where modellers specify simple actions to carry out based on a range of possible events. VEPL provides a more powerful, clearer and extensible event-handling language than this approach. VEPL filter/actions can handle more complex events, provide a wider range of actions, be extended by building hierarchical models (or using an API), and have graphical representations which more clearly show event flow in a process model. Most process modelling languages use textual rules with guard predicates and executable actions to specify how events are handled. ProcessWEAVER provides a textual co-shell language which allows users to specify actions for process model nodes when fired by input tokens. SLANG uses textual specifications of guards and actions for nodes in a state transition network. Marvel and Oz use guarded rules specified over data types.

Adele provides an activity manager, which uses a textual language to specify database-related event handling for process models. These approaches all use textual languages, and hence complex rules suffer from the same problems as when presenting other process model structures textually. The relationships between different stages, tools and artefacts are expressed in a linear, textual fashion, unlike graphical VEPL event handling models where event flows and process model artefacts are more clearly represented in graph form. This is a particular problem for rules which span multiple process stages, as understanding rule behaviour requires several textual data and task specifications to be viewed. VEPL permits specification of both simple and very complex event handling graphically, resulting in high-level visualisations of event handling behaviour. For example, the specification of a simple work coordination mechanism where a user is notified whenever another user modifies a particular work artefact, shown in Figure 11, results in an intuitive event handling model which is relatively easy for users to understand and modify. By allowing previously-defined filter/action templates to be composed using the same event propagation notation, VEPL supports reuse of a wide range of event handling behaviour. In contrast, approaches which use textual codification to handle these kinds of events, produce event handlers that are less clear and generally less reusable than those of VEPL. Visual dataflow-based languages, such as Fabrik (Ingalls et al., 1988) and Prograph (Cox et al., 1989), provide graphical dataflow models which are similar in nature to VEPL, but use dataflow rather than event-flow, which is less appropriate in a process modelling domain. Some visual languages, such as ViTABaL (Grundy and Hosking, 1995), utilise an event-driven model but lack the equivalent of Serendipity's filters, actions, and interest specification capabilities. Because of their general-purpose nature, these visual programming languages lack specific process modelling capabilities, and thus can not express and represent process model event-handling and work coordination tasks as effectively as EVPL and VEPL.

Most process modelling languages are supported by their own PCE or Workflow Management System (WFMS). For example E³ PML is supported by the E³ p-draw tool (Jaccheri and Gai, 1992), SLANG is supported by the SPADE PCE (Bandinelli et al., 1994; Bandinelli et al., 1993; Bandinelli et al., 1996), VPL by the Regatta (Swenson et al., 1994) and TeamFLOW (TeamWARE, 1996) WFMSs, and obligation nets by the wOrlds CSCW environment (Bogia and Kaplan, 1995). These environments typically allow process models to be enacted and advanced as users complete work on tasks and subtasks. Environments utilising textual languages are limited in how they can inform users of the state of enacted process models. Most avoid using the textual codification of the process model to visualise their state. In contrast, EVPL process models are animated to give users feedback on process model enactment and other users' work. Action Workflow, wOrlds, Regatta and TeamFlow also use highlighting of enacted process models. However in Serendipity, we have gone further than these systems, and highlight stages as they are enacted/deenacted, highlight stages of cooperating users, and highlight artefacts, tools and roles currently being used by an enacted process stage (Grundy et al., 1996b).

Most recent Computer Supported Cooperative Work (CSCW) research has focused on low-level interaction mechanisms, such as synchronous and asynchronous editing. Examples include most Groupware systems (Ellis et al., 1991), GroupKit (Roseman and Greenberg, 1996), Mjølner (Magnusson et al., 1993), C-MViews (Grundy et al., 1995c), and Rendezvous (Hill et al., 1994). These systems lack information about the "work context" changes have been carried out in, whereas Serendipity makes this information readily accessible to users. There is some work on providing higher-level process modelling and coordination facilities, such as workflow configuration (Medina-Mora et al., 1992), obligations (Kaplan et al., 1992b, Bogia and Kaplan, 1995), and shared workspace awareness (Roseman and Greenberg, 1996), but these systems are generally separate from the work artefacts or editing tools, and are not used to provide work context information. In Serendipity the text chats and note annotation CSCW tools incorporate information from the enacted process models, as do other tools integrated with Serendipity. This provides users with more context awareness capabilities than are supported by other CSCW and process modelling systems.

Oz (Ben-Shaul and Kaiser, 1994a) enables the definition of high-level work coordination capabilities, such as summits and treaties, utilising extensions to the Marvel rule-based model (Heineman et al., 1992),

and supports cooperative transactions for itself and integrated tools (such as ProcessWEAVER) via an external concurrency control architecture (Heineman and Kaiser, 1995). TeamFLOW, wOrlds and ConversationBuilder use obligations between process stages to enable enactment events to be propagated between process stages from different diagrams. ProcessWEAVER (Fernström, 1993) uses transition networks to model synchronisation of concurrent activities by cooperative agents, but like wOrlds these deal mainly with “enactment”-style events. VEPL supports the definition of complex work coordination by using EVPL constructs in its event-handling models. We have used VEPL to build a variety of low-level concurrency control mechanisms, typically supported by groupware tools. We have also used it to build higher-level work coordination models similar to those used by ProcessWEAVER and wOrlds (Grundy et al., 1996b). Serendipity uses both a shared server to broadcast events and share data, and a repository storing local process model and work artefacts. This allows it to support Oz-style summits, with process model enactment events generated by multiple users broadcast via the shared server. VEPL models can handle and constrain data sharing, in a similar manner to the “diplomats” utilised by Oz treaties, and can also support concurrent transactions by having actions store sequences of artefact changes and undo them if multiple transaction conflicts arise.

There have been several attempts at integrating CSCW, ISDEs and workflow/PCEs, including ConversationBuilder (Kaplan et al., 1992b), MultiviewMerlin (Marlin et al., 1993), wOrlds (Bogia and Kaplan, 1995), SPADE/ImagineDesk (Di Nitto and Fuggetta, 1995; Bandinelli et al., 1996), ProcessWEAVER and Oz (Heineman and Kaiser, 1995), and various groupware tools and Oz (Ben-Shaul et al., 1994b; Ben-Shaul and Kaiser, 1996). Many of these, such as MultiviewMerlin, SPADE/ImagineDesk and wOrlds, have produced environments with a high degree of integration. However none have produced environments which capture detailed information about enacted process models and present this “work context” information using the integrated tools themselves, nor do they allow information from the integrated tools to be utilised in the process modelling environment exactly as though the tools were part of that environment. In contrast in our integrated environment the boundaries of the two systems, from a user's perspective, have disappeared, despite there being no modifications to the underlying architectures of either system. The augmentation of SPE and CSCW tool data with Serendipity process model information, and the highlighting of in-use SPE artefacts by Serendipity, allows users to remain aware of when and why other users have made artefact changes or caused tool events to occur. Similarly, SPE artefacts (classes, diagrams etc.) and tools (OOA/D editor, text editor, debugger etc.) can be represented in Serendipity as EVPL artefacts and tools, and VEPL filter/actions can use these artefacts and tools as both producers of events and as event filters. Serendipity process stages can also store work histories using the event descriptions generated by the integrated tools. This provides closer user interface integration between process modelling tool, CSCW tools and work tools than do these other PCE, CSCW environment and work tool integration efforts.

Our integration of Serendipity, MViews CSCW tools, and an ISEE has been very successful, due to the event-driven and component-based natures of these environments and their common implementation platform. We have achieved a much lesser degree of integration of Serendipity with third-party tools, because of limitations in the interfaces provided to the tool data and control functions. Recent work on integrating such heterogeneous software development, CSCW and process modelling environments has focused on data aspects (e.g. federated approaches to tool integration (Bounab and Godart, 1995)), control integration via enveloping (e.g. between Oz and third party tools (Valetto and Kaiser, 1995)), and process integration (e.g. coordinating tool usage via PCEs (Marlin et al., 1993; Di Nitto and Fuggetta, 1995; Bandinelli et al., 1996)). Serendipity provides, via MViews and VEPL, an event-driven interface which supports control and process integration with disparate tools. At present, Serendipity utilises a repository which stores process model data in the form of Prolog terms. While Serendipity actions can be defined to facilitate import/export of this data with heterogeneous tools, this does involve more work than utilised in federated approaches to tool integration via distributed data management (Bounab and Godart, 1995; Valetto and Kaiser, 1995).

8.3. CURRENT AND FUTURE WORK WITH SERENDIPITY

The experience of the authors and other users of Serendipity to date has been generally positive, but a number of desirable improvements to the languages and environment have been identified. The EVPL and VEPL languages have proved concise and yet expressive for a large range of process modelling and work coordination/event-handling problems. There is a need, however, to support better artefact structuring and repository querying facilities for users. To this end we are currently developing a third visual notation for Serendipity, the Visual Query Language (VQL). This will allow users to specify the structure of artefacts using an MViews-style component and relationship model. Users will also be able to define complex, graphical queries over MViews (and other tool) environment repositories. Triggering of filters and actions will then be possible over the query result, which will be incrementally updated as data is modified.

We have been working on filters and actions which allow work contexts to be determined dynamically, in addition to being specified using artefact, role and tool representations. This will allow Serendipity to determine changes of context (i.e. change of current enacted process stage) automatically, rather than requiring users to do this manually, as at present. Actions will utilise the history of work context information for stages to determine such context changes. Visualisation techniques allowing users to view summaries of inter-related changes made in different contexts are also being developed. These facilities will, we believe, assist Serendipity in better supporting more informal aspects of work, which is currently poorly supported by existing workflow and PCEs (Kaplan et al., 1996).

We are currently porting Serendipity to Java and are utilising a Web-based interface to the modelling language views (Grundy et al., 1997). This will improve the portability of our environment. More significantly, this will enable us to provide better integration with heterogeneous tools, or at least utilise the now commonly-understood “plug-in” and “helper application” model of loose integration employed by browser-style applications. Another advantage will be the ability to reuse other people’s CSCW and work tools, in the form of Java applets, plug-ins or helper applications. We have found that systems built on event-based architectures, such as those of MViews, are much more amenable to tool integration, and plan to use Serendipity’s event-handling language in conjunction with such systems to better facilitate their integration with Serendipity.

9. Summary

Our work with Serendipity makes a number of new contributions to research into process technology. EVPL, while having some similarities to other graphical process modelling languages, utilises a versatile enactment event-based execution model. EVPL is used for generic process modelling, meta-process modelling and detailed work planning, and supports process stage work context modelling via representations of artefacts, tools, role communication and various kinds of inter-component usage relationships. VEPL is a novel event-handling notation which utilises EVPL model components, as well as introducing filters and actions, to handle enactment, artefact update, tool-induced and role-induced events. EVPL and VEPL together allow process modellers to specify arbitrary event-handling for process models, which includes specifying a large variety of work coordination strategies, automatic process model rule application, and propagation and storage of events between stages, artefacts, tools and roles.

Serendipity provides a support environment for EVPL and VEPL which includes multiple views of process models and event handling specifications. Enactment of process models utilises highlighting of enacted stages, event flows and artefact, tool and role representations, helping multiple users to keep aware of others’ work contexts. Collaborative editing of process models is supported, along with CSCW tools for annotating, messaging and talking about models. Integration of Serendipity and MViews-based tools for performing work results in environments with highly integrated user interfaces. Descriptions of changes made in other tools are annotated with work context (i.e. process stage) information, process

stages store lists of changes made while they are enacted (forming histories of work), and icons in both Serendipity and the integrated tools are highlighted to facilitate group awareness. Such tight user interface integration is not supported by most other PCE/workflow and work tool integration efforts. Serendipity utilises a component, event-based architecture. A hybrid technique is used to store process model information in local repositories for speed of access, and a central repository used to enable sharing of versions and broadcasting of enactment, artefact and tool events.

Serendipity is being extended to incorporate a flexible Visual Query Language, which will double as a tool repository query and data visualisation language, and a more flexible artefact data specification for use by VEPL filters and actions. We are currently porting MViews to Java and will also port Serendipity and the MViews suite of software development tools, to improve their accessibility, performance, and ability to integrate existing tools. Filters and actions are currently interpreted, but will be compiled to Java to improve their performance.

Acknowledgments

The authors gratefully acknowledge the many helpful comments of the anonymous reviewers on earlier drafts of this paper.

References

- Baldi, M., Gai, S., Jaccheri, M.L., and Lago, P. 1994. Object Oriented Software Process Design in E³. In *Software Process Modelling & Technology*, eds. A. Finkelstein and J. Kramer and B. Nuseibeh, Research Studies Press, 1994.
- Bandinelli, S., Fuggetta, A., and Ghezzi, C. 1993. Process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, vol. 19, no. 12, pp. 1128-1144.
- Bandinelli, S., Fuggetta, A., Ghezzi, C., and Lavazza, L. 1994. SPADE: an environment for software process analysis, design and enactment. In *Software Process Modelling & Technology*, eds. A. Finkelstein and J. Kramer and B. Nuseibeh, Research Studies Press, 1994.
- Bandinelli, S., Di Nitto, E., and Fuggetta, A. 1996. Supporting cooperation in the SPADE-1 environment, *IEEE Transactions on Software Engineering*, vol. 22, no. 12.
- Barghouti, N.S. 1992. Supporting Cooperation in the Marvel Process-Centred SDE. In *Proceedings of the 1992 ACM Symposium on Software Development Environments*, ACM Press, 1992, pp. 21-31.
- Belkhatir, N., Estublier, J., and Melo, W.L. 1994. The Adele/Tempo Experience. In *Software Process Modelling & Technology*, eds. A. Finkelstein and J. Kramer and B. Nuseibeh, Research Studies Press, 1994.
- Ben-Shaul, I.Z. and Kaiser, G.E. 1994a. A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment. In *Sixteenth International Conference on Software Engineering*, IEEE CS Press, pp. 179-188.
- Ben-Shaul, I.Z., Heineman, G.T., Popovich, S.S., Skopp, P.D. and Tong, A.Z., and Valetto, G. 1994b. Integrating Groupware and Process Technologies in the Oz Environment. In *9th International Software Process Workshop: The Role of Humans in the Process*, Ghezzi, C., IEEE CS Press, Airlie, VA, October 1994, pp. 114-116.
- Ben-Shaul, I.Z. and Kaiser, G.E. 1996. Integrating Groupware Activities into Workflow Management Systems. In *7th Israeli Conference on Computer Based Systems and Software Engineering*, Tel Aviv, Israel, June 1996, pp. 140-149.
- Bogia, D.P. and Kaplan, S.M., Flexibility and Control for Dynamic Workflows in the wOrlds Environment. In *Proceedings of the Conference on Organisational Computing Systems*, ACM Press, Milpitas, CA, November 1995.
- Bounab, M. and Godart, C. 1995. A Federated Approach to Tool Integration. In *Proceedings of CAiSE'95*, Springer-Verlag, LNCS 932, Finland, June 13-16 1995, pp. 269-282.
- Conradi, R., Hagaseth, M., Larsen, J., Nguyen, M.N., Munch, B.P., Westby, P.H., Zhu, W., Liu, C. 1994. *EPOS: Object Oriented Cooperative Process Modeling*. In *Software Process Modelling & Technology*, eds. A. Finkelstein and J. Kramer and B. Nuseibeh, Research Studies Press, 1994.
- Cox, P.T., Giles, F.R., and Pietrzykowski, T. 1989. Prograph: a step towards liberating programming from textual conditioning. , IEEE Computer Society Press. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, 1989, pp. 150-156.
- Di Nitto, E. and Fuggetta, A. 1995. Integrating process technology and CSCW. In *Proceedings of IV European Workshop on Software Process Technology*, LNCS, Springer-Verlag, Leiden, The Netherlands, April 1995.
- Ellis, C.A., Gibbs, S.J., and Rein, G.L. 1991. Groupware: Some Issues and Experiences, *Communications of the ACM*, vol. 34, no. 1, 38-58, January 1991.
- Fernström, C. 1993. ProcessWEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.

- Grundy, J.C. and Hosking, J.G. 1993. A framework for building visual programming environments. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, 1993, pp. 220-224.
- Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. 1995a. Coordinating, capturing and presenting work contexts in CSCW systems. In *Proceedings of OZCHI'95*, Wollongong, Australia, Nov 28-30 1995, pp. 146-151.
- Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. 1995b. Connecting the pieces, In *Visual Object-Oriented Programming*, eds. M. Burnett, A. Goldberg, T. Lewis, Manning/Prentice-Hall, 1995.
- the 6th European Workshop on Next Generation of CASE Tools, Finland, June 1995, pp. 109-116.
- Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Amor, R. 1995c. Support for Collaborative. Integrated Software Development. In *Proceeding of the 7th Conference on Software Engineering Environments*, IEEE CS Press, April 5-7 1995, pp. 84-94.
- Grundy, J.C. and Hosking, J.G. 1995. ViTABaL: A Visual Language Supporting Design By Tool Abstraction. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, IEEE CS Press, Darmstadt, Germany, September 1995, pp. 53-60.
- Grundy, J.C. and Venable, J.R. 1995a. Providing Integrated Support for Multiple Development Notations. In *Proceedings of CAiSE'95*, Springer-Verlag, LNCS 932, Finland, June 1995, pp. 255-268.
- Grundy, J.C., and Venable, J.R. 1995b. Developing CASE tools that support integrated design notations. In *Proceedings of Grundy, J.C., Hosking, J.G., and Mugridge, W.B. 1996a. Towards a Unified Event-based Software Architecture. In Joint Proceedings of the SIGSOFT'96 Workshops*, eds. L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, ACM Press, October 14-15 1996, pp. 121-125.
- Grundy, J.C., Hosking, J.G., and Mugridge, W.B. 1996b. Low-level and high-level CSCW in the Serendipity process modelling environment. In *Proceedings of OZCHI'96*, IEEE CS Press, Hamilton, New Zealand, Nov 24-27 1996.
- Grundy, J.C., Venable, J.R., Hosking, J.G., and Mugridge, W.B. 1996c. Coordinating collaborative work in an integrated Information Systems engineering environment. In *Proceedings of the 7th Workshop on the Next Generation of CASE tools*, Crete, 20-21 May 1996.
- Grundy, J.C., Hosking, J.G., and Mugridge, W.B. 1996d) Supporting flexible consistency management via discrete change description propagation, *Software - Practice & Experience*, vol. 26, no. 9, 1053-1083, September 1996.
- Grundy, J.C., Mugridge, W.B., and Hosking, J.G. 1996e. A Java-based toolkit for the construction of multi-view editing systems. In *Proceedings of the Second Component Users Conference*, Munich, July 14-18 1997.
- Grundy, J.C. and Hosking, J.G. 1996. Constructing Integrated Software Development Environments with MViews, *International Journal of Applied Software Technology*, vol. 2, no. 3-4, pp. 133-160.
- Grundy, J.C., Hosking, J.G., and Mugridge, W.B. 1997. A Visual, Java-based Componentware Environment for Constructing Multi-view Editing Systems, In *Proceedings of 2nd Component Users Conference*, Munich, July 1997.
- Harmsen, F., and Brinkkemper, S., 1995. Design and Implementation of a Method Base Management System for a Situational CASE Environment. In *Proceedings of the 2nd Asia-Pacific Software Engineering Conference*, IEEE CS Press, Brisbane, December 1995, pp. 430-438.
- Heineman, G.T., Kaiser, G.E., Barghouti, N.S., and Ben-Shaul, I.Z. 1992. Rule Chaining in Marvel: Dynamic Binding of Parameters, *IEEE Expert*, vol. 7, no. 6, 26-32, December 1992.
- Heineman, G.T. and Kaiser, G.E. 1995. An Architecture for Integrating Concurrency Control into Environment Frameworks. In *Proceedings of the 17th International Conference on Software Engineering*, IEEE CS Press, Seattle, Washington, April 1995, pp. 305-313.
- Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F., and Wilner, W. 1994. The Rendezvous Architecture and Language for Constructing Multi-User Applications, *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 2, June 1994, 85-125.
- Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., and Doyle, K. 1988. Fabrik: A Visual Programming Environment. In *Proceedings of OOPSLA '88*, ACM Press, pp. 176-189.
- Jaccheri, M.L. and , Gai, S. 1992. Initial Requirements for E3: an Environment for Experimenting and Evolving Software Processes. In *Proceedings of EWSPT'92*, Trondheim, Norway, September 1992, pp. 99-102.
- Jaccheri, M.L. and Conradi, R. 1993. Techniques for Process Model Evolution in EPOS, *IEEE Transaction on Software Engineering: Special issue on Software Process Evolution*, vol. 19, no. 12, 1145-1156, December 1993.
- Kaplan, S.M., Tolone, W.J., Carroll, A.M., Bogia, D.P., and Bignoli, C. 1992a. Supporting Collaborative Software Development with ConversationBuilder. In *Proceedings of the 1992 ACM Symposium on Software Development Environments*, ACM Press, pp. 11-20.
- Kaplan, S.M., Tolone, W.J., Bogia, D.P., and Bignoli, C. 1992b. Flexible, Active Support for Collaborative Work with ConversationBuilder. In *1992 ACM Conference on Computer-Supported Cooperative Work*, ACM Press, pp. 378-385.
- Kaplan, S.M., Fitzpatrick, G., Mansfield, T., and Tolone, W.J. 1996. Shooting into Orbit. In *Proceedings of Oz-CSCW'96*, DSTC Technical Workshop Series, University of Queensland, Brisbane, Australia, August 1996, pp. 38-48.
- Kellner, M.I., Feiler, P.H., Finkelstein, A., Katayama, T., Osterweil, L.J., Penedo, M.H., and Rombach, H.D. 1990. Software Process Modelling Example Problem. In *Proceedings of the 6th International Software Process Workshop*, IEEE CS Press, Hokkaido, Japan, 28-31 October 1990.
- Krishnamurthy, B. and Hill, M. 1994. CSCW'94 Workshop to Explore Relationships between Research in Computer Supported Cooperative Work & Software Process. In *Proceedings of CSCW'94*, ACM Press, pp. 34-35.
- Magnusson, B., Asklund, U., and Minör, S. 1993. Fine-grained Revision Control for Collaborative Software Development . In *Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, Los Angeles CA, December 1993, pp. 7-10.

- Marlin, C., Peuschel, B., McCarthy, M., and Harvey, J. 1993. MultiView-Merlin: An Experiment in Tool Integration. In *Proceedings of the 6th Conference on Software Engineering Environments*, IEEE CS Press.
- Medina-Mora, R., Winograd, T., Flores, R., and F., Flores. 1992. The Action Workflow Approach to Workflow Management Technology. In *Proceedings of CSCW'92*, ACM Press, pp. 281-288.
- Roseman, M. and Greenberg, S. 1996. Building Real Time Groupware with GroupKit, A Groupware Toolkit, *ACM Transactions on Computer-Human Interaction*, Vol 3, No. 1, March 1996, 1-37.
- Swenson, K.D. 1993. A Visual Language to Describe Collaborative Work. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE CS Press, 1993, pp. 298-303.
- Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B., and Irwin, K. 1994. A Business Process Environment Supporting Collaborative Planning, *Journal of Collaborative Computing*, vol. 1, no. 1.
- TeamWARE, Inc. 1996. *TeamWARE Flow*, <http://www.teamware.us.com/products/flow/>.
- Tolone, W.J., Kaplan, S.M., and Fitzpatrick, G. 1995. Specifying DynamicSupport for Collaborative Work within wOrlds. In *Proceedings of the 1995 ACM Conference on Organizational Computing Systems*, Milpitas, CA, August 1995, pp. 55-65.
- Valetto, G. and Kaiser, G.E. 1995. Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments. In *IEEE Seventh International Workshop on Computer-Aided Software Engineering*, July 1995, pp. 40-48.
- Venable, J.R. and Grundy, J.C. 1995. Integrating and Supporting Entity Relationship and Object Role Models. In *Proceedings of the 14th Object-Oriented and Entity Relationship Modelling Conferece*, Springer-Verlag, LNCS 1021, Gold Coast, Australia, 1995.

5.2 A decentralized architecture for software process modeling and enactment

Grundy, J.C. Hosking, J.G., Mugridge, W.B., Apperley, M.D. A decentralised architecture for software process modelling and enactment, IEEE Internet Computing, Vol. 2, No. 5, September/October 1998, IEEE CS Press, pp. 53-62.

DOI: [10.1109/4236.722231](https://doi.org/10.1109/4236.722231)

Abstract: Many development teams, especially distributed teams, require process support to adequately coordinate their complex, distributed work practices. Process modeling and enactment tools have been developed to meet this requirement. The authors discuss the Serendipity-II process management environment which supports distributed process modeling and enactment for distributed software development projects. Serendipity-II is based on a decentralized architecture and uses Internet communication facilities.

My contribution: Developed initial ideas for the research, did majority of tool design, implemented and evaluated tool, wrote majority of the paper, investigator for funding for the project from FRST

A decentralized architecture for software process modeling and enactment

John C. Grundy[†], John G. Hosking^{††}, Warwick B. Mugridge^{††} and Mark D. Apperley[†]

[†]Department of Computer Science, University of Waikato
Private Bag 3105, Hamilton, New Zealand
{jgrundy, mapperle}@cs.waikato.ac.nz

^{††}Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john, rick}@cs.auckland.ac.nz

Abstract

Centralised client/server approaches to supporting software process modelling and enactment are common, but can suffer from serious robustness, performance and security problems. We describe a decentralised architecture for software process modelling and enactment, which also incorporates distributed work coordination, task automation and system integration facilities. Our environment based on this architecture uses visual, multiple view specifications of work processes, together with animation of process model views to support enactment awareness. This environment provides a robust, fast and secure system for coordinating work on distributed software development projects utilising basic internet communication facilities.

Keywords: process modelling and enactment, decentralised software architectures, computer-supported cooperative work, work coordination, distributed software agents

1. Introduction

Process modelling and enactment tools support cooperating workers in defining software process models which describe the way they work, or at least how they should work, on distributed software development projects [2, 15]. Running, or "enacting", these models allows developers, whose work may be distributed over time and place, to more easily follow prescribed or recommended processes and to more effectively coordinate their work [5, 8]. These process support tools may also include software "agents", used to automate simple tasks, support work coordination, integrate use of disparate tools, and track and record histories of software development work for future reference and analysis. Many development teams, especially distributed teams, require such support to adequately coordinate their complex, distributed work practices.

Most existing process support tools, and most cooperative work supporting tools in general, use centralised, client-server software architectures [2, 15], and often can only be used with dedicated local area networks. A central server permits collaborative process modelling, and a centralised process enactment engine runs ("enacts") these process models, records work, and supplies task automation and systems integration agents. However, research has shown that such centralised architectures can have serious robustness, performance, security and distribution problems [1, 6]. Attempts have been made to provide distributed process enactment engines, to overcome some of these problems, and some work has been done in facilitating distributed task automation [1, 6]. However, little work has been done to facilitate distributed work process modelling and more generalised software agents.

We have developed a distributed architecture and environment for fully decentralised software process modelling, process enactment and distributed work coordination, task automation and tool integration agents. Our architecture provides significantly improved robustness, performance, security and distribution capabilities over centralised process support tools. It can be run over local area networks, internet wide area networks, and modem and mobile computer networks, and supports a wider range of distributed collaborative process modelling and software agents than most other distributed process support systems.

2. The Serendipity-II Process Management Environment

We have been developing process modelling and enactment environments for defining work processes for a variety of professional domains, including software engineering, patent law, real estate, and immigration consultancy, and a

variety of office automation tasks, including inventory management and project management [8]. We developed the decentralised Serendipity-II process modelling environment for use on these tasks, an improved version of an earlier, central-server based process support tool, Serendipity [8]. Serendipity-II provides multiple views of process models using a range of mainly visual languages. Figure 1 illustrates the basic process modelling and enactment capabilities of Serendipity-II. The left view shows a simple software process model for modifying a software system. Ovals represent process stages, and include a unique name and role performing the work. Enactment flows, which may be labelled, connect stages and pass on "enactment events", driving process model execution.

Stages may have subprocess models defined, such as the bottom right view of Figure 1, which shows the subprocess for the "2. Design changes" process stage. Input and output event icons, shown as hexagons, indicate where enactment flows enter and leave subprocesses, and constrain the process stage these enactment events flow into or out from. Role assignment views, such as the one in the top right view, allow process modellers to associate particular users, or software agents, with stage roles. Other views supported include usage of tools and artefacts by process model stages and complex artefact structure definitions. The dialogue at centre top shows the enacted stages for each user (here only "john", the project leader, is doing work for this process model - planning specific tasks for his co-workers).

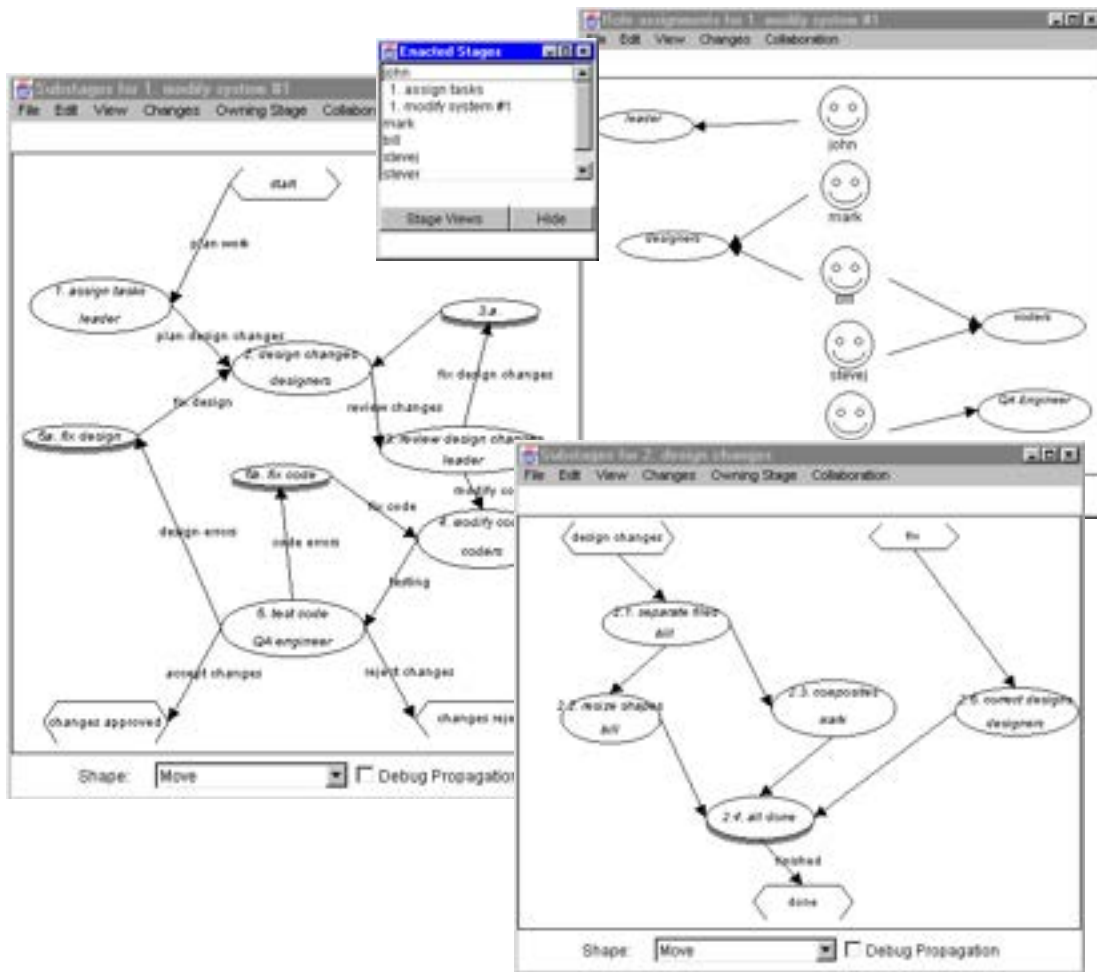


Figure 1. An example of process modelling and enactment in Serendipity-II.

In order to provide robust, efficient, secure and distributed process support using Serendipity-II's capabilities, we required an architecture for this environment that would satisfy the following requirements:

- Users are able to collaboratively edit process models both synchronously (i.e. what-you-see-is-what-I-see style) and asynchronously (i.e. editing alternate versions with subsequent version merging). These collaborative editing, and supporting communication capabilities (audio, text chat, messaged, annotation etc.) should be decentralised to ensure robustness and efficient performance.
- Users are able to enact process models in a decentralised way i.e. have their own process enactment engines. Various awareness capabilities should enable users to track each other's work e.g. which process stages they are currently working on, which others are enacted, histories of enactments and work, etc.

- Decentralised work coordination agents should be supported. Local agents should not need to access other users' process models and enactment information, and agents coordinating multiple users should communicate in a decentralised way (running either on a particular user's machine or as independent "environments").
- Users must have full control over access by others to their process models, and the deployment of software agents which affect their work.
- Process modelling, enactment and work coordination agent facilities must work equally well over both high- and low-bandwidth network connections, and must be tolerant of periodic disconnection from the network.

3. A Decentralised Process Management Architecture

We have implemented the Serendipity-II process modelling and enactment using the JComposer metaCASE tool and the JViews object-oriented framework [9]. JViews is implemented in Java and provides a component-based software architecture for building multi-view, multi-user environments, extending the Java Beans componentware API. JComposer provides visual languages supporting the specification of tool repository and view components, and the specification of editor icon appearance and functionality. JComposer generates JViews class specialisations, with tool developers able to further extend these generated classes to code complex functionality in Java.

Figure 2 shows the basic process model information for Serendipity-II being specified in JComposer. JComposer provides basic abstractions of "components" (rectangular icons), relationships (ovals), attributes and links (labelled arcs). Base (i.e. repository) information for Serendipity-II includes stages and enactment links, together with the tool repository itself and a hashtable relationship linking all base stages to the repository. Multiple views of this repository can be specified, adding additional information about the process modelling language, as well as definitions of multiple representations (views) of the repository information [9]. JViews classes implementing Serendipity-II process modelling capabilities are generated from these JComposer specifications. We have extended these generated classes to incorporate process enactment capabilities by hand-coding using Java.

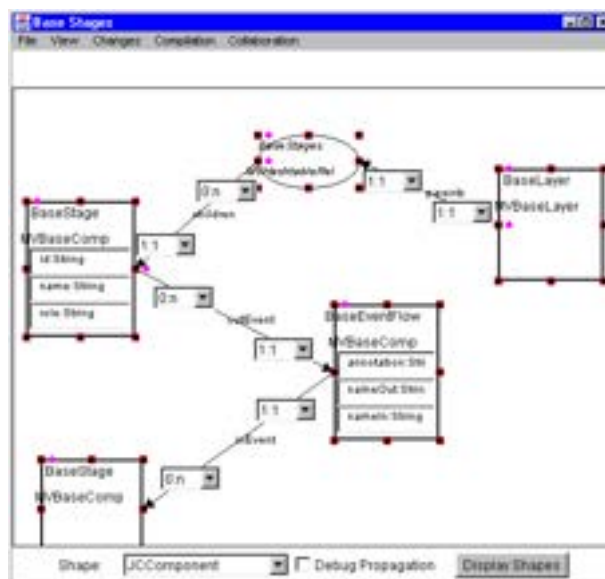


Figure 2. Simple Serendipity-II repository components being specified in JComposer.

The generated JViews implementation of Serendipity-II represents process stage and enactment information as components, linked by relationships. When a component is modified, a "change description" object is generated, documenting this state change event. Change descriptions are propagated to relationships which pass the change event onto other components, carry out processing in response to the event (e.g. enforcing constraints or updating multiple views of a component), or ignore the change [7]. We have used this same change propagation mechanism to implement the process enactment engine of Serendipity-II, by representing enactment events as change descriptions and propagating these to connected components representing other process stages and links. JViews change descriptions can also be stored in "version records" to facilitate work tracking, undo/redo, versioning of components via deltas, and be broadcast to other users' JViews-based environments to facilitate cooperative work.

We have used JViews' component and change description serialisation and inter-environment propagation mechanisms to develop a decentralised software architecture for Serendipity-II, eliminating centralised servers. Every user's environment is responsible for; its own communication with other environments; its own process model enactment; and

storing its own process model and enactment components. It may also have its own "local" software agents. Figure 3 illustrates this architecture.

Each user of Serendipity-II has a modelling and enactment environment that provides multiple views of work processes with which users interact. Each environment has its own "receiver" (server) and "sender" (broadcasting client) components, used to communicate with other users' environments. Thus rather than a centralised server, communication is via one user to zero or more other users. In large systems, communication can be from user to "groups" of other users i.e. forwarding agents are used to propagate information to groups of other users and/or distributed software agents. "Users" in our architecture may also include Serendipity-II, or third-party, software agents and Information Systems interfaces. Our architecture treats these in the same manner as components that interface to people. Both sender and receiver run asynchronously with editing and software agent processes, ensuring fast response time for user editing and good agent processing performance over both high- or low-bandwidth networks.

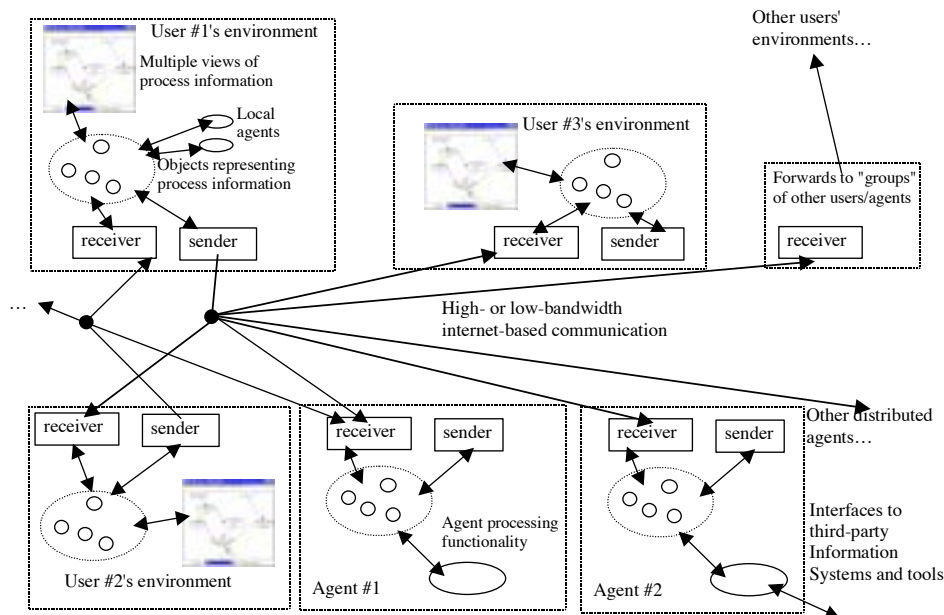


Figure 3. A decentralised process modelling and enactment architecture.

Component objects in our JViews architecture have system-allocated unique component identifiers. Each user is given an "identifier factory", ensuring all components created by a user's environment have a unique id for their own as well as other users' environments. Components also have an identifier indicating the component they have been copied from (if any), allowing simple mapping between components representing different versions of the same information to be performed. Environments exchange components, groups of components and change descriptions by using the JViews component and change description serialisation mechanisms. Serialised model and enactment information is sent to other users' environments by the user's sender component (currently using TCP-IP sockets). Receiver components either map the deserialised component and change description (event) component identifiers to components in their own environment, or create new copies for any components they do not have copies of.

Thus in our architecture, some components may be local to a user's environment, representing private process information. All shared process model information, however, is fully replicated between users' environments, using our JViews component identification and versioning scheme. We use this fully replicated architecture so that shared process information can be independently modified even if one or more users sharing the information are off-line, mobile or their network connections are temporarily lost. This also allows a seamless transition at any time between synchronous and asynchronous process model view editing.

It is possible to keep parts of a shared process model synchronised, if required. This is achieved by incrementally propagating change descriptions generated by components representing shared information between environments. When received, the component identifiers of these change descriptions are mapped to corresponding components in the receivers' environments, and these components are updated to conform to the changes made by other users. Alternatively, a modified set of components can be copied to other users' environments, replacing the old versions. Sets of change descriptions representing changes made to parts of a shared process model by other users can also be incrementally merged with a user's existing model. These change descriptions are sent to the other user's environment, and the other user then requests the changes be actioned on the components in their environment representing their copy

of the shared process model. We have a change object annotation facility which allows a receiving environment to detect the loss of change objects and to request they be resent if necessary.

Our component-based implementation has a further advantage in that interfacing to third-party tools, information systems, and software agents can be done using Java Beans' component interface mechanisms, or by providing reusable JViews components to interface to these systems. These interfaces can be used by software agents defined in Serendipity-II to facilitate heterogeneous tool integration.

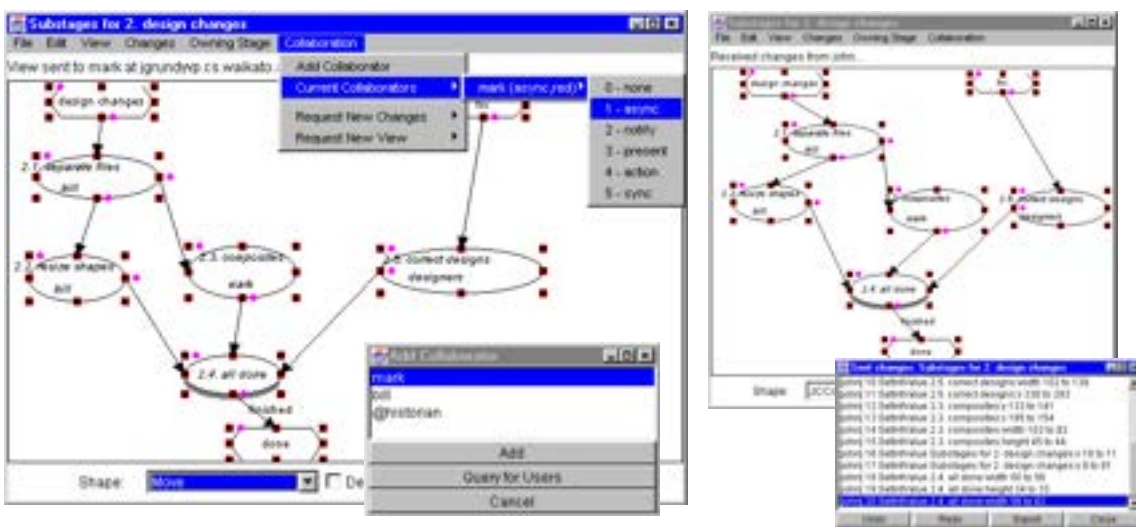
The following sections illustrate how we have used this decentralised architecture to provide a range of distributed process modelling and enactment facilities, and distributed work coordination, task automation and work tracking agents, over the internet in a decentralised fashion.

4. Collaborative Process Modelling

Serendipity-II supports decentralised process modelling. Users decide whether to collaboratively edit process specifications synchronously, or to independently model and evolve processes asynchronously, subsequently merging changes. Such temporary divergence is useful when evolving individual subprocess models. The ability to evolve process models when some users are off-line, using mobile computers or their internet connections temporarily lost is also increasingly important; the constraints of centralised process modelling interfere with these needs.

Collaborative process modelling and software agent specification are managed by broadcasting whole process model view definitions, or incremental changes to model views, between users' environments. A receiving user's environment maps changes to objects in the sending environment's view to objects in the receiving environment's view. The unique object copy identifier tags are used to map changes between alternate versions of the same object in the two environments. Asynchronous and synchronous editing differ in when changes are mapped.

Figure 4 shows an example of asynchronous process modelling. Users John and Mark have independently modified their versions of the process model shown. To reconcile their alternative versions, they can exchange whole copies of the modified view and manually reconcile them. Alternatively, they can exchange a list of change event objects (each describing an incremental change) which have been stored with the view. Each may then incrementally merge selected changes with their own version of the process model (as shown in the dialogue). This allows users to, for example, tailor aspects of a shared process model view (such as appearance and layout), or have different stages/enactment flows in their process model while it evolves. We have found this partial sharing approach to be useful during process model evolution. It is also useful if a particular user encounters exceptions in a process model which they wish to correct for themselves, but not have the effect immediately impact on others users' versions of the same subprocess. An additional advantage is one of control over one's own process definitions; users view and approve selected changes rather than have them always forced upon them by others' work.



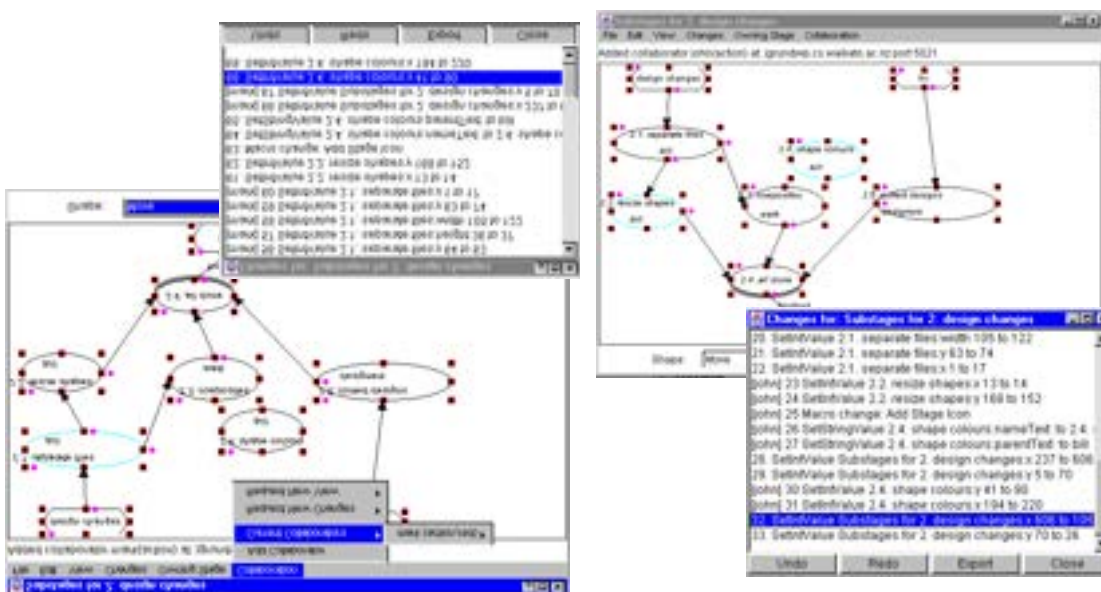
(a) John's view

(b) Mark's View

Figure 4. Asynchronous process modelling example.

At times users desire closer collaboration over process modelling. In addition to asynchronous editing, Serendipity-II supports a range of "closer" editing modes ranging through to fully synchronous. *Present* mode allows users to be informed of changes being made to shared process model views by other collaborators as each change occurs. A description of each change is added to a dialogue, and users can choose to immediately merge the presented changes with their own process view, delay merging, or negotiate with other users about the change. In *action* mode all received changes are automatically applied (i.e. incrementally merged) as they occur. *Synchronous* mode similarly applies all changes immediately, but provides a locking mechanism ensuring there are no simultaneous updates by multiple users of the same view component. Locks are obtained by broadcasting a message to all users synchronously editing the view and obtaining a lock for the requesting user for the component to be edited from each environment. If two users simultaneously request a lock, it is denied to both and they must retry their edit, or coordinate their editing using chat, audio or other communication support.

Colouring of process model view icons is used to indicate the last user to change an icon in the view. Figure 5 shows users John and Mark synchronously editing a process model view. The changes in the view modification history are annotated with the name of the user who made each change.



(a) John's view

(b) Mark's view

Figure 5. Synchronous process modelling example.

Our process modelling architecture is robust against users going off-line during collaborative editing. When they later return, their environment requests any changes made during their absence to be forwarded to them. They can then review and optionally merge these changes into their affected process model views. Our architecture also ensures ownership and visibility of views is in the control of the view creator, who decides who may be sent a copy of their views (using the "Add Collaborator" option in the Colloration menu, shown in Figure 5). Receiving users also decide whether or not they want their repository of process information updated to conform to a new view they have been sent. Fast semi-synchronous and synchronous collaborative editing performance is ensured as changes to view objects are only propagated to those other users (or distributed agents) interested in the changes. The asynchronous threading of the sender components for a user's environment broadcasts changes "in the background", ensuring view editing response for the user is maximised.

5. Distributed Process Enactment

A process enactment engine is included with each user's environment. Users may enact a process model stage (i.e. run it), finish a stage (say its complete), suspend or terminate a stage, or indicate the stage they are currently working on (which we call the "current enacted stage"). Each of these enactment activities generates an enactment events. These events may cause other stages to be enacted, finished etc. For example, finishing a stage in a particular "finishing state" enacts those connected stages that are specified to be enacted when the stage ends in this state.

Enactment events are propagated to other users' environments, informing them of the stage enactment and ensuring their process models are enacted appropriately i.e. flow-on effects of enactments are actioned in other users' environments. Enactment events are, however, only propagated to other users who have an "interest" in them, ie they have a copy of the subprocess enacted and have some role assignment in that subprocess. This reduces unnecessary enactment event propagation, and keeps enactment events for local subprocesses "private" i.e. prevents monitoring of users' "private" work.

Figure 6 shows a simple example of an enacted process model. John has finished planning tasks (stage "1. assign tasks"), and a finished enactment event has flowed into the "2. design changes" stage. This has resulted in the "2.1. separate files" stage being enacted for Bill. The stages in the process model views have been coloured to indicate enacted stages and the currently enacted stage for each user. An "enactment monitor" dialogue (bottom right) shows enacted stages i.e. assigned work for all users who have roles in the overall process model being used. The '*' indicates the currently enacted stage for a user. An enactment history dialogue (bottom left) shows the history of enactment events for stage "2. design changes".

Our decentralised process model enactment approach has significant advantages over centralised approaches. Failure of any user's environment or a central server does not prevent other users from modelling and enacting their process models. As enactment events are recorded by environments, users rejoining the network can query these events, in the same manner they query for changes to process model views. In addition, large numbers of enactment events, which can occur if many users or many automated tasks are used, tend not to greatly reduce performance of our system as they are broadcast asynchronously and only to other relevant users.

One interesting consequence of our environment permitting asynchronous editing of process model views is that these can be inconsistent between different users' environments i.e. users may try to enact different versions of the same process model. This turns out to be generally unproblematic, as although environments may receive enactment events from engines running different versions of process models, these events may still be mapped onto the user's existing version stages, or shown to users if they cannot be mapped. We have found this approach allows users more freedom to evolve tailor the shared process models for their own purposes while still allowing them to monitor other users' work.

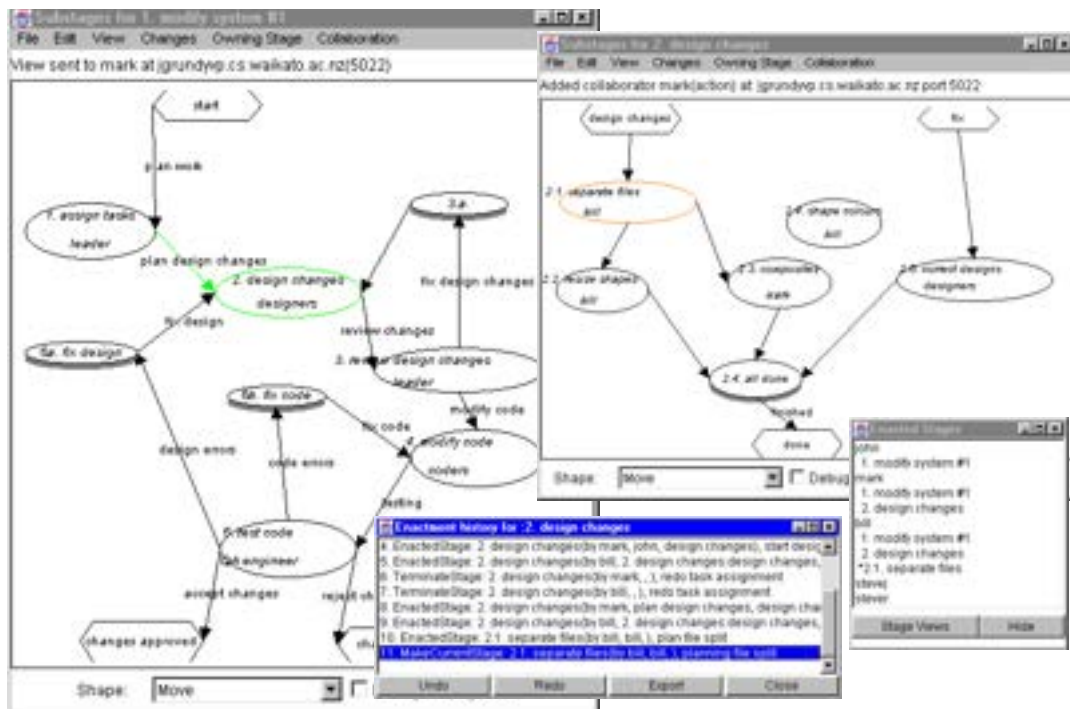


Figure 6. A simple example of an enacted process model.

6. Work Coordination Agents

While process modelling and enactment capabilities provide useful work coordination tools, they are by themselves of insufficient flexibility and power to assist users in managing complex cooperative software development work [8]. To address this, we have developed a novel visual language to support the specification of a range of software agents,

including task automation, work history tracking, work coordination and tool integration, and to build extensions to the basic Serendipity-II process model behaviour. This event-based visual language comprises event "filters" (rectangular icons with a "?" prefix) and "actions" (oval, shaded icons).

Users connect event filters to process stages, roles, artefacts or tools, and these filters pass on any enactment, communication, artefact update or tool events matching specified criteria. Actions receive events, usually from filters, and perform specified processing in response. Figure 7 shows two simple examples of software agents specified by filters and actions. The model on the left detects when a stage is enacted or finished, and runs an action to download or upload artefacts relevant to the stage from a shared file server. The "request stage artefacts" and "put back stage artefacts" actions connect to the file server using sockets to transfer files (artefacts) associated in other views with the "2. design changes" stage. These actions are examples of packaged interfaces to third-party Information Systems (in this case a shared file server) being integrated with the process enactment engine. The model on the right defines an agent which detects changes made to a work artefact and stores these changes in an event history artefact (represented by the plain rectangular icon). Filter and action models can be packaged and parameterised by inputs and outputs, like subprocess models, and reused in different process model specifications.

One implication of our decentralised Serendipity-II architecture is that software agents can be run: locally for a user's environment only; in another user's environment; or by autonomous agents with their own "environment" (sender/receiver and uniquely identified object pool). Each approach is appropriate for different kinds of agent. For example, running agents locally is useful for basic task automation.

We have built agents to: add functionality to Serendipity-II, such as time delay enactment and automatic enactment; store selected changes to artefacts and process models; store selected stage enactment events; and interface to third-party systems. The latter include tools for software development (editors, compilers and CASE tools), office automation (word processors, spreadsheets, and databases) and communication (email, chat and audio).

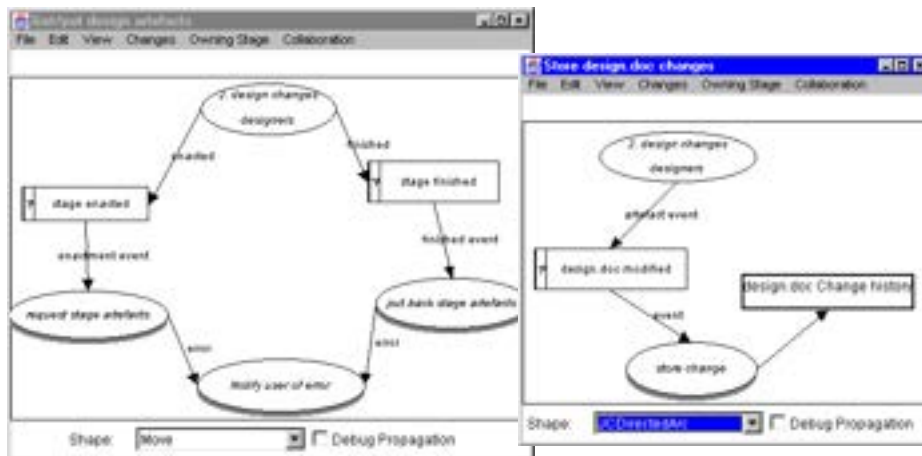


Figure 7. Two simple software agents specified in Serendipity-II.

Sometimes it is necessary for agents to process events generated in other users' environments. Such software agents must either request other environments send them such events (by asking other users to allow "event forwarding" agents to be added to their environments), or these software agents must be run entirely by the other users' environments. To achieve the latter, users specify the event processing they want with Serendipity-II filter and action views, then send these views to other users, who decide whether or not to allow the software agents specified in them to run in their environment. This allows users to ensure inappropriate agents which could adversely affect their work are not run within their environments. This was a major drawback with our original centralised Serendipity architecture: users lacked sufficient control over the agents others could specify which affected their work [8].

Autonomous agents are specified in the same way as other agents, and are sent to "robot" environments, which don't have a user interface but instead only run automatic processing. These robots run filter/action models sent to them, usually communicating with multiple users' environments to request appropriate events to process. These agents may continue to run when a user disconnects from the network. On reconnection, they can be queried for events that occurred during a user's absence or for results of processing. We have built several such autonomous agents, for example to provide a centralised work history tracking and querying facility, to embody constraints which affect multiple users' process models, and for additional group awareness capabilities. Our decentralised architecture can tolerate such agents failing and being restarted, whereas most process support systems using centralised approaches can not.

Figure 8 shows an example of a distributed software agent being specified by user "John". The left hand view is sent to user "Bill's" environment to forward changes of interest to John automatically. Bill must however first concur to have the components of this view added to his Serendipity-II repository. The right-hand view specifies that changes forwarded from Bill's environment are stored for later perusal by John (but these could, for example, be passed onto further filters and actions for automatic processing).

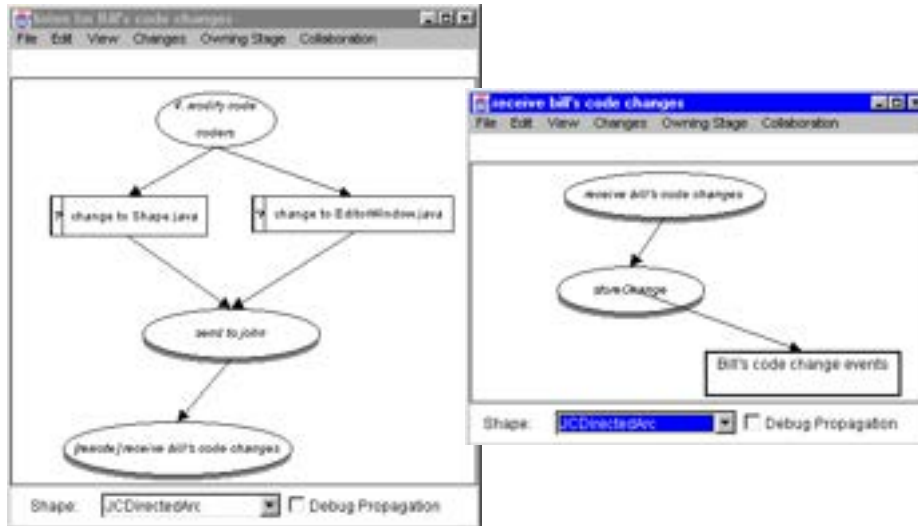


Figure 8. An example of work coordination using distributed filters and actions.

7. Discussion

Many WFMS based on centralised, client-server style architectures, such as Regatta [15], Action Workflow [13], SPADE [2], and ProcessWEAVER [5] suffer from performance problems when database access and message passing frequencies are high. This is particularly problematic where large numbers of developers need to coordinate work, and where many autonomous work coordination and task automation agents are employed. Such systems also suffer notorious robustness problems, with the central server being their weakest link [6]. In contrast, our decentralised architecture performs well even under heavy loading. This is due to the devolution of responsibility for inter-environment communication to each user's environment, with no central server bottlenecks. Users' environments communicate only with "interested" environments when collaborative process modelling. When enacting process models, initial enactment events are sent to others' environments, whose own enactment engines then action these events. Robustness is ensured as any user or agent can be removed from the network and the rest of the system still continues to function, with dynamic rerouting of messages (if necessary) possible. When a user's environment reestablishes its connections to the network, other users and agents are queried for changes to components and events that occurred when it was off-line. Our environments record all such events to ensure this is possible.

Various other decentralised approaches supporting workflow enactment have been developed, including those of METUFlow [6], Exotica/FMQM [1], and MOBILE [10]. Some of these, such as MOBILE and Exotica, use replicated servers, process migration and assignment of parts of workflows to specific servers. These require complex algorithms to assign workflow processing, and still retain servers, which may fail or become bottlenecks. METUFlow assigns workflow execution to CORBA objects, with computed guards controlling distributed execution, but requires a block-structured workflow language and a guard construction mechanism. Many of these approaches are not tolerant of periodic network disconnection for neither process modelling nor enactment. While much work has been done on distributed workflow enactment, most such systems do not support decentralised specifications of workflow models, but require client-server or single-user workflow definition [1, 13, 15], like most other collaborative editing tools. We have found our decentralised approach to both workflow modelling and enactment to be more flexible and robust. An additional advantage of Serendipity-II is its use of visual, event-based workflow modelling, enactment and software agent specification languages. Such languages tend to be more readily understandable by end-users than many textual, rule-based languages [5, 8, 15].

Various Computer-supported cooperative work (CSCW) systems have been developed to facilitate collaborative modelling and design. Examples include those with centralised architectures, such as GROVE [4] and ImagineDesk [3]. Those with decentralised architectures include GroupKit [14], Orbit [11] and Mjolner [12]. Generally, CSCW tools with

centralised architectures share the same performance and robustness limitations as centralised process support tools. Many decentralised CSCW systems, like GroupKit, also do not scale up for use on large, multiple view modelling domains, such as process modelling. This is often due to their adoption of universal synchronous or asynchronous editing modes for all users on different kinds of views. In Serendipity-II, we allow users to decide on the appropriate editing mode for each view, and which collaborators have access to the view and thus need be sent editing events. This minimises unnecessary editing event propagation.

Performance, robustness, security and distribution of Serendipity-II for process modelling and enactment has been very good in the software process modelling and office automation domains we have used it for. We have been deploying the environment on several small office automation process modelling applications, with some users on a local area network and others using mobile computers. We have also deployed it on a medium-size software process modelling and enactment application, with seven software developers and a variety of tools being used in conjunction with Serendipity-II. These include program editors and compilers, communication and file management services and JComposer being used as an OOA/D tool. Most developers are using a local area network, but two work from home using modem connections, and two are at another location 100km away. The internet provides a seamless, unifying communication mechanism between all environments enabling all developers to coordinate their work using Serendipity-II. Useability studies of Serendipity-II are underway and will help us further enhance the environment, and we plan to deploy it in other process modelling domains.

8. Summary

We have described the Serendipity-II process modelling and enactment environment and its decentralised architecture. This architecture uses multiple point-to-point communication across the internet between distributed users' and software agents, obviating the need for centralised servers. This results in more robust, efficient, secure and easily distributable process modelling and enactment and decentralisation of task automation and work coordination agents. Distributed process modelling allows users to join and leave a network of cooperating process modellers, with both asynchronous and synchronous process model view editing supported. Distributed process enactment is supported by broadcasting process enactment events between related users' environments, once again allowing users to join or leave the network of users and tolerating failure of any node in this network. Serendipity-II provides a visual process modelling language and a visual event-based software agent definition language. Software agents defined using event filters and actions can be run locally, or sent to be run in other users' environments, or by autonomous agent environments.

We are currently building further software agents to assist users in managing the complexities of distributed work. These include agents supporting document sharing, revision histories, interfaces to legacy development tools, and improved awareness of other users' work. Providing facilities allowing users to specify groups of other users to forward objects and events to, and to specify autonomous agent environment configurations, will allow end users of Serendipity-II to more easily configure their decentralised process support systems.

References

- [1] Alonso, G., Mohan, C., Gunthor, R. and Agrawal, D., ElAbadi, A., and Kamath, M., "Exotica/FMQM: A Persistent Message-based Architecture for Distributed Workflow Management," in *Proceedings of the IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organisations*, Norway, 1995.
- [2] Bandinelli, S., DiNitto, E., and Fuggetta, A., "Supporting cooperation in the SPADE-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, no. 12.
- [3] DiNitto, E. and Fuggetta, A., "Integrating process technology and CSCW," in *Proceedings of IV European Workshop on Software Process Technology*, Lecture Notes in Computer Science, Springer-Verlag, Leiden, The Netherlands, April 1995.
- [4] Ellis, C.A., Gibbs, S.J., and Rein, G.L., "Groupware: Some Issues and Experiences," *Communications of the ACM*, vol. 34, no. 1, 38-58, January 1991.
- [5] Fernström, C., "ProcessWEAVER: Adding process support to UNIX," in *2nd International Conference on the Software Process: Continuous Software Process Improvement*, IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.
- [6] Gokkoca, E., Altinel, M., Cingil, I., Tatbul, E.N., Koksals, P., and Dogac, A., "Design and implementation of a Distributed Workflow Enactment Service," in *Proceedings of 2nd IFCIS Conference on Cooperative Information Systems*, Charleston, SC, USA, June 1997.
- [7] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," *Software - Practice & Experience*, vol. 26, no. 9, 1053-1083, September 1996.
- [8] Grundy, J.C. and Hosking, J.G., "Serendipity: integrated environment support for process modelling, enactment and work coordination," *Automated Software Engineering*, vol. 5, no. 1, January 1998.

- [9] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Static and Dynamic Visualisation of Software Architectures for Component-based Systems," in *Proceedings of SEKE'98*, KSI Press, San Francisco, June 18-20 1998.
- [10] Heintz, P. and Schuster, H., "Towards a Highly Scaleable Architecture for Workflow Management Systems," in *Proceedings of the 7th International Conference and Workshop on Database and Expert Systems*, Zurich, Switzerland, September 1996.
- [11] Kaplan, S.M., Fitzpatrick, G., Mansfield, T., and Tolone, W.J., "Shooting into Orbit," in *Proceedings of Oz-CSCW'96*, DSTC Technical Workshop Series, University of Queensland, Brisbane, Australia, August 1996, pp. 38-48.
- [12] Magnusson, B., Asklund, U., and Minör, S., "Fine-grained Revision Control for Collaborative Software Development," in *Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, Los Angeles CA, December 1993, pp. 7-10.
- [13] Medina-Mora, R., Winograd, T., Flores, R., and Flores, F., "The Action Workflow Approach to Workflow Management Technology," in *Proceedings of CSCW'92*, ACM Press, 1992, pp. 281-288.
- [14] Roseman, M. and Greenberg, S., "Building Real Time Groupware with GroupKit, A Groupware Toolkit," *ACM Transactions on Computer-Human Interaction*, vol. 3, no. 1, 1-37, March 1996.
- [15] Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B., and Irwin, K., "A Business Process Environment Supporting Collaborative Planning," *Journal of Collaborative Computing*, vol. 1, no. 1.

5.3 Collaboration-Based Cloud Computing Security Management Framework

Almorsy, M., Grundy, J.C. and Ibrahim, A. Collaboration-Based Cloud Computing Security Management Framework, In *Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD 2011)*, Washington DC, USA on 4 July – 9 July, 2011, IEEE, pp. 364-371.

DOI: [10.1109/CLOUD.2011.9](https://doi.org/10.1109/CLOUD.2011.9)

Abstract: Although the cloud computing model is considered to be a very promising internet-based computing platform, it results in a loss of security control over the cloud-hosted assets. This is due to the outsourcing of enterprise IT assets hosted on third-party cloud computing platforms. Moreover, the lack of security constraints in the Service Level Agreements between the cloud providers and consumers results in a loss of trust as well. Obtaining a security certificate such as ISO 27000 or NIST-FISMA would help cloud providers improve consumers trust in their cloud platforms' security. However, such standards are still far from covering the full complexity of the cloud computing model. We introduce a new cloud security management framework based on aligning the FISMA standard to fit with the cloud computing model, enabling cloud providers and consumers to be security certified. Our framework is based on improving collaboration between cloud providers, service providers and service consumers in managing the security of the cloud platform and the hosted services. It is built on top of a number of security standards that assist in automating the security management process. We have developed a proof of concept of our framework using .NET and deployed it on a test bed cloud platform. We evaluated the framework by managing the security of a multi-tenant SaaS application exemplar.

My contribution: Developed initial ideas for the research, co-supervised the two PhD students, wrote substantial parts of paper

Collaboration-Based Cloud Computing Security Management Framework

Mohemed Almarsy, John Grundy and Amani S. Ibrahim

Computer Science & Software Engineering, Faculty of Information & Communication Technologies
Swinburne University of Technology, Hawthorn, Victoria, Australia
{malmorsy, jgrundy, aibrahim}@swin.edu.au

Abstract — Although the cloud computing model is considered to be a very promising internet-based computing platform, it results in a loss of security control over the cloud-hosted assets. This is due to the outsourcing of enterprise IT assets hosted on third-party cloud computing platforms. Moreover, the lack of security constraints in the Service Level Agreements between the cloud providers and consumers results in a loss of trust as well. Obtaining a security certificate such as ISO 27000 or NIST-FISMA would help cloud providers improve consumers trust in their cloud platforms' security. However, such standards are still far from covering the full complexity of the cloud computing model. We introduce a new cloud security management framework based on aligning the FISMA standard to fit with the cloud computing model, enabling cloud providers and consumers to be security certified. Our framework is based on improving collaboration between cloud providers, service providers and service consumers in managing the security of the cloud platform and the hosted services. It is built on top of a number of security standards that assist in automating the security management process. We have developed a proof of concept of our framework using .NET and deployed it on a testbed cloud platform. We evaluated the framework by managing the security of a multi-tenant SaaS application exemplar.

Keywords: cloud computing; cloud computing security; cloud computing security management

I. INTRODUCTION

The cloud computing model represents a new paradigm shift in internet-based services that delivers highly scalable distributed computing platforms in which computational resources are offered 'as a service'. Although the cloud model is designed to reap uncountable benefits for all cloud stakeholders including cloud providers (CPs), cloud consumers (CCs), and service providers (SPs), the model still has a number of open issues that impact its credibility.

Security is considered one of the top ranked open issues in adopting the cloud computing model, as reported by IDC [1]. A reasonable justification of such increasing concerns of the CCs about cloud security [2] includes: (1) The loss of control over cloud hosted assets (CCs become not able to maintain their Security Management Process (SMP) on the cloud hosted IT assets); (2) The lack of security guarantees in the SLAs between the CPs and CCs; and (3) the sharing of resources with competitors or malicious users. Accordingly, no matter how strongly the model is secured, consumers continue suffering from the loss of control and lack of trust problems. On the other hand, the CPs struggle with the cloud platform security issues because the cloud model is very complex and has a lot of dimensions that must be considered when developing a holistic security model [2]

including the complex architecture of the cloud model, the model characteristics, the long dependency stack, and the different stakeholders' security needs. These dimensions result in a large number of heterogeneous security controls that must be consistently managed. Moreover, the CPs host services they are not always aware of the contents or the security requirements to be enforced on these services. This leads to a loss of security control over these services and the cloud platforms.

Although much research into cloud services security engineering has been undertaken, most efforts focus only on the cloud based services offered, such as web services. Such efforts have investigated capturing security requirements and generating corresponding WS-Security configurations. However, they pay no attention to the underlying platform security or the other cloud service delivery models such as IaaS and SaaS. They also do not address the impact of the multi-tenancy feature introduced by the cloud model on the security of the cloud delivered services.

Two new community projects are trying to tackle the CCs trust problem by introducing a list of best practices and checklists such as CSA - GRC project [3], or by aligning existing security standards to the cloud model such as FedRAMP [4]. Both projects' focus is to obtain CCs trust by assessing and authorizing the cloud platforms. These projects lack the consumers' involvement in specifying their security requirements and managing their SMP. The later project fits better with CPs deliver their own services only.

In this paper we introduce a novel approach that tackles both loss of trust and security control problems by enabling CCs to extend their SMP to include cloud hosted assets. Our approach introduces a new cloud security management framework based on aligning the NIST-FISMA standard [12], as one of the main security management standards, to fit with the cloud architectural model. The information required to put the NIST standard into effect is not possessed by one party. Thus we improve the collaboration among the key cloud stakeholders to share such required information. Getting CCs involved in every step of the SMP of their assets mitigates claims of loosing trust and control. Our approach also mitigates the loss of control claimed by the CPs for the hosted services that are developed by other parties. Being based on a security management standard our approach enables both parties to get security certifications. Our approach helps stakeholders to address the following issues:

- What are the security requirements needed to protect a cloud hosted service given that the service is used by different tenants at the same time?

- What are the appropriate security controls that mitigate the service adoption risks and who select such controls?
 - Are the selected controls available on the cloud platform or we will/can use third party controls?
- What are the security metrics required to measure the security status of our cloud-hosted services?

To validate our approach we developed a prototype of our collaboration-based cloud security management framework and deployed it on a cloud platform hosting a SaaS application (an ERP Service). We evaluated the approach by securing the ERP service assuming that the cloud platform has multiple tenants sharing the same cloud application. Each tenant has their own security requirements and SMP.

In section II we use a motivating scenario to highlight the research problems we aim to address. Then we give an overview of cloud computing security issues and the SMP. Section III reviews related work in cloud computing security research areas. Section IV discusses our approach and security standards used. Section V describes our framework architecture. Section VI explains a usage example of the developed framework. In Section VII we discuss the implications of our work and further research.

II. MOTIVATION

A. A Motivating Example

Swinburne University is going to purchase a new Enterprise Resource Planning (ERP) solution in order to improve its internal process. After investigation, Swinburne decided to adopt the Galactic ERP solution (a cloud-based solution), to save upfront hardware investment required and to optimize infrastructure costs. Galactic is a Web-based solution developed by SWINSOFT. SWINSOFT hosts its applications on a cloud platform delivered by GREENCLOUD. GC delivers IaaS and PaaS. SWINSOFT uses third party services to accelerate the development process. Such services are developed by GC and deployed on the GC platform including: (1) Workflow-Builder service (customizable workflow management service), (2) Currency-Now service (retrieves the current exchange rate of currencies), (3) Batch-MPRD (used in posting operations based on the map-reduce model). At the same time, Auckland University has the same interest in using the Galactic ERP solution, as shown in Figure 1. Swinburne and Auckland are security certified. *Swinburne* needs to maintain a similar security level on Galactic as applied on their internal IT systems. *Auckland* assigns high risk impact to the Galactic asset. Thus each stakeholder has different security constraints to enforce on the same service.

B. The Cloud computing model security problem

The cloud model has different dimensions that participate in complicating its security problem including [2]:

1) The model has different Service Delivery Models (SDMs): Infrastructure as Service (IaaS), Platform as Service (PaaS), and Software as Service (SaaS). Each SDM has different possible implementations (SaaS may be hosted

on top of PaaS or IaaS) and its own security issues based on the underlying technology. Accordingly each SDM has a set of security controls that are required to mitigate such issues.

2) The cloud model has two key characteristics: *Multitenancy* which results in virtualizing the boundaries among the hosted services of different tenants, and *elasticity* which requires secure services' migration and placement strategies.

3) The model has a long stack of dependent layers where the security of each layer depends on lower layers' security.

4) The model has different stakeholders involved including CPs, SPs, and CCs. Each stakeholder has their own security needs that may conflict with other stakeholders' needs.

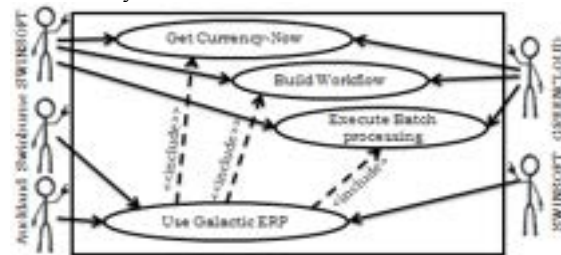


Figure 1: A use case diagram for the motivating example

C. Information Security Management Systems

Information security management systems (ISMS) are defined in ISO27000 as [6] “systems that provide a model for establishing, implementing, operating, monitoring, reviewing, maintaining and improving the protection of information assets.”. These operations are grouped into three main phases:

1) Defining security requirements - this phase includes (i) identifying security goals/objectives that the ISMS should satisfy and deliver, (ii) conducting risk analysis and assessment to identify existing risks within the system scope, and (iii) detailing objectives/risks into detailed security requirements and security policies.

2) Enforcing security requirements - this phase includes: (i) identifying security controls to be used, and (ii) implementing and configuring such controls based on the specified security requirements.

3) Monitoring and Improving security - this phase includes (i) monitoring the current status of the implemented security controls, (ii) analysing the measured security status to identify existing security issues, and (iii) maintaining and improving the current security controls.

D. Key challenges

After analyzing the cloud computing model security problem, the ISMS process, and the motivating scenario we have identified the following key problems:

- 1) Each stakeholder has their own SMP that they want to maintain/extend to the cloud hosted assets.
- 2) No stakeholder can individually maintain the whole security process of the cloud services because none of them has the full information required to manage security and each one has a different perspective.

- 3) Multi-tenancy requires maintaining different security profiles for each tenant on the same service instance.
- 4) No Security SLA is available that can be used to maintain agreements related to cloud assets security.
- 5) The existing standards such as ISO27000 and FISMA do not map well to the cloud model because these standards consider the SMP from the platform/asset owner not from a Service Provider perspective.

E. Key requirements of the cloud ISMS

Any proposed security management framework for the cloud model should cover the following key requirements:

- 1) Enable CCs to specify their security requirements on the cloud hosted assets and the underlying cloud platform.
- 2) Enable CCs to monitor their assets security status and the underlying platform security status as well.
- 3) Support for multi-tenancy where different tenants can maintain their SMP with strong isolation of data.
- 4) Be based on existing security management standards that are already adhered by the CCs and CPs.

III. RELATED WORK

A. Cloud security engineering

Menzel et al [7, 8] proposed a model driven approach and a language to specify security requirements on web services and cloud web applications composed of web services. Each application instance (and its services) is deployed on a VM. They assumed that (1) web applications are composed of web services only, (2) multi-tenant security is maintained through using VMs for each tenant (simplest case), and (3) the underlying infrastructure security is not considered. Bertram et al [9] proposed a similar idea of security engineering for cloud hosted services with more higher level of abstraction (risk-based instead of security-requirements-based). The authors assumed a trusted and secured cloud platform with a focus to provide security PaaS that can manage and mitigate security risks of the services shared among two collaborating enterprises. Both efforts cover only Web services and capture/generate security on the service level without considering the underlying layers.

B. Cloud security management

Saripalli et al [10] proposed a quantitative risk analysis and assessment method based on NIST- FIPS-199 [5]. Risk assessment is a step in the SMP. The remaining steps of the SMP are still required. Although the authors proposed a quantitative method in assessing risks, they used qualitative evaluation bands (Low, Medium, and High). Similar efforts were carried out by Xuan et al [11]. ISO27000 [6], NIST-FISMA [12] are the two main ISMS standards. Both standards do not fit well with the cloud model because they assume that asset owner has full control over the SMP of his assets (hosted inside enterprise boundaries). Moreover, they do not consider the scenario of sharing a service “Multi-tenancy” among consumers. Related research efforts in ISMS include risk assessment and management frameworks

such as OCTAVE [13], CORAS [14], Security management systems such as policy-based security management [15], Ontology-based and policy based management has been merged in one approach [16], and model-based security management [17]. Most of these approaches focus on the security capturing and enforcement phases rather than the feedback and improvement phases of the SMP. These phases become more critical in the cloud model because we moved from security within enterprise boundaries to securing assets hosted on third-party platforms.

C. Cloud Security SLA management

Security SLA is another approach to specify and manage security. Although a lot of proposals have been introduced in SLA management (SLA specification, enforcement and monitoring), security is rarely considered as it’s different from the other QOS attributes such as performance and reliability. Shirlei et al [18] focused on Sec-SLA objectives related to data backup policy only. Pankesh et al [19] has proposed a cloud SLA management architecture but security is not covered. A reasonable justification of the lack of Sec-SLAs is the difficulty in defining suitable security metrics.

IV. OUR APPROACH

Our approach is based on improving and supporting collaboration among cloud stakeholders to develop a cloud security specification and enforcement covering all of their needs. Our approach is based on aligning FISMA standard with the cloud model and utilizing collaboration among the stakeholders to maintain a cloud security specification covering their needs. We first illustrate how we aligned the FISMA standard to fit with the cloud computing model.

A. Aligning NIST-FISMA standard with the cloud model

The Federal Information Security Management Act (FISMA) standard [20] defines a framework for managing the security of information and information systems that support the operations of the agencies. The framework has six main phases including: service security categorization, security controls selection, security controls implementation, security controls assessment, service authorization, and security monitoring. Table 1 summarizes how we aligned FISMA model to fit with the cloud model.

(1) Service Security Categorization - Each service (S_j) on the cloud platform can be used by different tenants. Each service tenant (T_i) owns their information only the shared service (S_j). The tenant is the only entity that can decide/change the impact of a loss of confidentiality, integrity and availability on their business objectives. Each tenant may assign different impact levels (Low, Medium, or High) to security breaches of their information. In FedRAMP [4], the CP specifies the security categorization of services delivered on their cloud platform. However, this is not sufficient as the CP does not have sufficient knowledge about the impact of information security breaches on their tenants’ business objectives.

Table 1: Alignment of NIST-FISMA standard with the cloud computing model

Phase	Task	CP	SP	CC	Inputs	Outputs
Security categorization	Categorize security impact (SC)	Informed	Informed	Responsible	Business objectives	Security Impact Level
Security controls selection	Register security controls	Responsible	Responsible	Responsible	Control Datasheet	Security controls registry
	Generate security controls baseline	Responsible (Automated by the framework)			Service SC + Controls registry	Controls baseline + matching status
	Assess service risks	Responsible			Service + platform arch. + service CVEs + CWE	Service Vulnerabilities + Threats + Risks
	Tailor security baseline	Responsible			Baseline + Risk assessment	Security mgmt plan (Sec-SLA)
controls implementation	Implement security controls	Responsible			Security mgmt plan	Updated Security plan
Security Assessment	Define security metrics	Responsible	Informed	Responsible	Security objective	Security assessment plan
	Assess security status	Responsible (Automated by the framework)			Security assessment plan	assessment report
Service Authorization	Authorize service	Informed	Informed	Responsible	Security plan + assessment report	Service authorization
Security Monitoring	Monitor security status	Responsible (Automated by the framework)			Security assessment plan	Security status report

Our approach enables CCs to be involved in specifying the security categorization of their information. Moreover, our approach enables both scenarios where we can consider the security categorization (SC) per tenant or per service. The security categorization of the service is calculated as the maximum of all tenants' categorizations:

$$SC(T_i) = \{(confidentiality, impact), (integrity, impact), (availability, impact)\}, Impact \in \{Low, Medium, High\} \quad Eq. (1)$$

$$SC(S_j) = \{(Confidentiality, Max(\forall T_i (impact))), (Integrity, Max(\forall T_i (impact))), (Availability, Max(\forall T_i (impact)))\} \quad Eq. (2)$$

(2) Security Control Selection - The selection of the security controls to be implemented to protect such assets from being breached has two steps: (a) baseline security controls selection. The FISMA standard provides a catalogue of security control templates categorized into three baselines (low, medium and high). Based on the security categorization of the tenant or the service we can select the initial baseline of controls that are expected to provide the required level of security specified by tenants; (b) Tailoring of the security controls baseline. We tailor the security controls baseline identified to cover the service possible vulnerabilities, threats, risks and the other environmental factors as follows:

I. The service risk assessment process

- *Vulnerabilities Identification* - this step requires being aware of the service and the operational environment architecture. We consider the involvement of the SP who knows the internal structure of the provided service and the CP who knows the cloud platform architecture.
- *Threat Identification* - the possible threats, threat sources and capabilities on a given service can be identified by collaboration among the SPs, CPs, and CCs. CCs are involved as they have the knowledge about their assets' value and know who may be a source of security breaches.

- *Risk Likelihood* - based on the capabilities of the threat sources and the nature of the existing vulnerabilities, the risk likelihood is rated as low, medium or high.
- *Risk Level (Risk Exposure)* - based on the risk impact (as defined in phase 1) and risk likelihood we drive the risk level as (Risk Level = Impact X Likelihood).

II. The security controls baseline tailoring process

Based on the risk assessment process, the selected security controls baseline is tailored to mitigate the new risks and to fit with the new environment conditions as follows:

- *Scoping of the Security Controls*: (i) Identify the common security controls; The cloud stakeholders decide on which security controls in the baseline they plan to replace with a common security control (either provided by the CPs or by the CCs), (ii) Identify critical and non-critical system components; the SPs and CCs should define which components are critical to enforce security on it and which are non-critical (may be because they are already in a trusted zone) so no possible security breaches, and (iii) Identify technology and environment related security controls that are used whenever required such as wireless network security controls.
 - *Compensating Security Controls* - whenever the stakeholders find that one or more of the security controls in the tailored baseline do not fit with their environment conditions or are not available, they may decide to replace such controls with a compensating control.
 - *Set Security controls parameters* - the last step in the baseline tailoring process is the security controls' parameters configuration, such as minimum password length, max number of unsuccessful logins, etc. This is done by collaboration between the CPs and CCs.
- The outcome of this phase is a security management plan that documents service security categorization, risks, vulnerabilities, and the tailored security controls baseline.

(3) Security Controls Implementation - The security plan for each tenant describes the security controls to be

implemented by each involved stakeholder based on the security control category (common, service specific). The common security controls implementation is the responsibility of the common control provider who may be the CPs (in case of internal security controls) or the CC (in case of external controls). The service-specific security controls implementation is the responsibility of the SPs. Each stakeholder must document the security controls implementation configurations in the security mgmt plan.

(4) Security Controls Assessment - Security controls assessment is required to make sure that the security controls implemented are functioning properly and meet the security objectives specified. This step includes developing a security assessment plan that defines what are the controls to be assessed, what are the assessment methods to be used, and what are the security metrics for each security control. The results of the assessment process are documented in a security assessment report. This step may result in going back to the previous steps in case of deficiency in the controls implemented or continuing with the next steps.

(5) Service Authorization - This step represents the formal acceptance of the stakeholders on the identified risks involved in the adoption of the service and the agreed on mitigations. The security plan and security assessment plan are the security SLA among the involved parties.

(6) Monitoring the Effectiveness of Security Controls - The CPs should provide security monitoring tools to help the CCs in monitoring the security status of their assets. The monitoring tools should have the capability to capture the required security metrics and report the collected measures in a security status report either event-based or periodic-based. The results of the monitoring process may require re-entering the SMP to handle new unanticipated changes.

B. Security automation

After aligning the FISMA standard with the cloud model we adopted a set of security standards to help improving the framework automation and its integration with the existing security capabilities, as shown in Figure 2 and Table 2.

Common Platform Enumeration (CPE) [21] -The CPE provides a structured naming schema for IT systems including hardware, operating systems and applications. We use the CPE as the naming convention of the cloud platform components and services. This helps in sharing the same service name with other cloud platforms and with the existing vulnerabilities databases - NVD [22].

Common Weakness Enumeration (CWE) and Common Attack Pattern Enumeration and Classification (CAPEC) [21] - The CWE Provides a catalogue of the community recognized software weaknesses. The CAPEC provides a catalogue of the common attack patterns. Each attack pattern provides a description of the attack scenario, likelihood, knowledge required and possible mitigations. We use the CWE and CAPEC as a reference for the cloud stakeholders during the vulnerabilities identification phase.

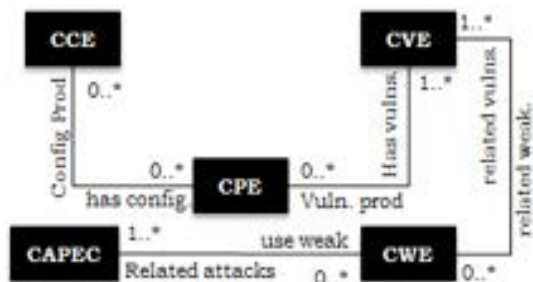


Figure 2: A class diagram of the adopted security standards

Common Vulnerability and Exposure (CVE) [21] - The CVE provides a dictionary of the common vulnerabilities with a reference to the set of the vulnerable products (encoded in the CPE). It also offers vulnerability scoring that reflects the severity of the vulnerability. We use the CVE to retrieve the know vulnerabilities discovered in the service or the platform under investigation.

Common Configuration Enumeration (CCE) [21] - The CCE provides a structured and unique naming to systems' configuration statements so that systems can communicate and understand such configurations. We use the CCE in the security controls implementation phase. Instead of configuring security controls manually, the administrators can assign values to security control templates' parameters. Our framework uses these configurations in managing the selected security controls.

Table 2: Formats of the adopted security standards

Standard	Format	Example
CPE	cpe/{part} : {vendor} : {product} : {version} : {update} : {edition} : {language}	cpe:/a:SWINSOFT:Galactic:1.0:update1:pro:en-us
CVE	CVE-Year-SerialNumber	CVE-2010-0249
CWE	CWE-SerialNumber	CWE-441
CAPEC	CAPEC-SerialNumber	CAPEC-113
CCE	CCE-softwareID-SerialNumber	CCE-17743-6

V. CLOUD SECURITY FRAMEWORK ARCHITECTURE

Our framework architecture consists of three main layers: a management layer, an enforcement layer, and a feedback layer. These layers, shown in Figure 3, represent the realization of the ISMS phases described in section II.

Management layer. This layer is responsible for capturing security specifications of the CPs, SPs, and CCs. It consists of: (a) The security categorization service used by the hosted services' tenants to specify security categorization of their information maintained by the cloud services; (b) The collaborative risk assessment service where all the cloud platform stakeholders participate in the risk assessment process with the knowledge they possess. (c) The security controls manager service is used to register security controls, their mappings to the FISMA security controls' templates, and their log files structure and locations. (d) The security metrics manager service is used by the cloud stakeholders to register security metrics they need to measure about the platform security. (e) The multi-tenant

security plan (SLA) viewer service is used to reflect the tenant security agreement. This shows the tenant-service security categorization, vulnerabilities, threats, risks, the selected mitigation controls and the required metrics. (f) The multi-tenant security status viewer. This reflects the current values of the security metrics and their trends.



Figure 3: The collaboration-based framework architecture

Enforcement layer. This layer is responsible for security planning and security controls selection based on the identified risks. The selected security controls are documented in the security management plan. The implementation service then uses this plan for maintaining security control configuration parameters and the mapping of such parameters to the corresponding security controls.

Feedback layer. This layer has two key services: the monitoring service which is responsible for collecting measures defined in the security metrics manager and storing it in the security management repository to be used by the analysis service and by the multi-tenant security status reporting service. The analysis service analyses the collected measures to make sure that the system is operating within the defined boundaries for each metric. If there is a deviation from the predefined limits, the analysis service will give alerts to update the current configurations.

VI. USAGE EXAMPLE

To demonstrate the capabilities of our cloud computing security framework and our prototype tool implementing this framework we revisit the motivating example from section II, a cloud based ERP system “Galactic” used by Swinburne and Auckland (CCs), developed by SWINSOFT (SP), and deployed on the GC (CP). The two tenants using the Galactic ERP services, Swinburne and Auckland, are still concerned about their assets’ security on the cloud. Both have their own SMP and their own security requirements to be enforced on their cloud assets.

The first step in our approach is to register the Galactic ERP service in the cloud platform service repository so that it can

be used by the CCs. This step can be done either by SWINSOFT or by the GC. In this step we use the CPE name as the service ID, Figure 4 (top). A new tenant, Auckland, can register their interest in using the Galactic service. Then Auckland will be granted a permission to manage the security of his information maintained by Galactic service. The same is done by Swinburne, Figure 4 (bottom).



Figure 4: Registering a service (top) and tenants (bottom)

Now Auckland and Swinburne can use our framework to maintain their SMP on their assets as follows:

1) **Service Security Categorization:** The Swinburne security administrator specifies the impact level of losing the confidentiality, integrity, and availability of their data maintained by the Galactic ERP service. The same will be done by the Auckland security administrator, as shown in Figure 4 (bottom). Whenever a new tenant registers their interest in a service and defines their security categorization of data processed by the service (or any of the existing tenants update his security categorization), the framework will update the overall service security categorization.

2) **Security Controls Selection:** The GC as a cloud provider already publishes their security controls database. Swinburne and Auckland can register their own security controls using the security controls manager service. Based on the security categorization step, the framework generates the security controls’ templates baseline. This baseline identifies the security controls’ templates that are: **satisfied** (matches one of the registered security controls), **missing** (does not match registered security controls), and **duplicate** (more than one matched control), shown in Figure 5.

a. **The Service Risk Assessment Process.** Galactic vulnerabilities are identified for the first time by SWINSOFT with the help of GC who know the architecture of the service and the hosting cloud platform. Both SWINSOFT and GC have the responsibility to maintain the service vulnerabilities list up to date. The framework enables to synchronize the service vulnerabilities with the community vulnerabilities database - NVD. Each CC – Swinburne and Auckland – should review the defined threats and risks on Galactic and append any missing

threats. The framework integrates with the CWE and CAPEC databases to help stakeholders in identifying possible vulnerabilities whenever the service does not have vulnerabilities recorded in the NVD.

#	CC Family	CC No.	Enhancement	CC Name	Control Status
Edit Delete	AC-	14	1		Missing
Edit Delete	AC-	17	1	Authenticator	Available
Edit Delete	AC-	17	1	SwinAntiVirus	Duplicate
Edit Delete	AC-	17	2	Authenticator	Available
Edit Delete	AC-	17	2	SwinAntiVirus	Duplicate

Figure 5: Security controls baseline with controls' status

#	Registration Date	Registration (Mths)	Security Categorization
1	1/01/2011	24	Low

Vulnerability Name	Vulnerability Description
CVE-2005-0413	Multiple SQL injection vulnerabilities in MyPHP Forum 1.0 allow remote attackers to execute arbitrary
CVE-2005-2473	getparam in netpbm does not properly use the "-dRAR" option when calling ghostscript to convert a
CVE-2005-4195	Multiple SQL injection vulnerabilities in Scout Portal Toolkit (SPT) 1.3.1 and earlier allow remote

Threat Name	Threat Description	Threat Source
DenialSvc	Denial of service	Attacker
InfoCopy	Copy of information at storage	Internal
InfoMod	Modification of information while being transferred	Attacker
InfoMod	Modification of data being processed	Malware

Risk Name	Risk Probability	Confidentiality Impact	Availability Impact	Integrity Impact	Risk Level
CC5	0.7	Low	High	Low	Medium

Control Name	Control Description	Control Baseline	Control Type	Control Family
Authenticator	an authentication security control	Low	Specific	Access Control
SwinAntiVirus	an antivirus security solution	Low	Common	System and Information Integrity
SwdIPS	an intrusion prevention system	Low	Common/Control	System and Information Integrity

Measurement Name	Measurement Description	Frequency	Measurement Status	Security Control
LoginActivity	Identify the user login rates	40	count(logstatus)	Authenticator

Figure 6: Auckland security management plan

b. The controls baseline tailoring process. The CCs decide which security controls in the baseline they plan to replace with common security controls provided by the CP or the CC, as shown in Figure 5. Then SWINSOFT, Auckland, and Swinburne select the critical service components that must be secured. Swinburne and Auckland define their security controls' parameter configurations. The security controls provided by the cloud platform can only be reviewed.

The final outcome of this step is a security management plan that documents the service security categorization, vulnerabilities, threats, risks, and the tailored security controls to mitigate the identified possible security breaches, as shown in Figure 6.

3) Security Controls Implementation: Each stakeholder implements the security controls under their responsibility as stated in the security plan and the security controls configurations as specified in the previous step.

4) Assessing the implemented security controls: The controls to be assessed and the objectives of the assessment are defined by GC, Auckland and Swinburne and documented in the tenant security assessment plan. The execution of such plan, the assessment process, should be conducted by a third party. Our framework helps in

assessing security controls status when using security controls that integrate with our framework (the framework can understand and read their log structure). The outcome of the assessment phase is a security assessment report.

5) Service Authorization: Swinburne and Auckland give their formal acceptance of the security plan, assessment plan, and the assessment reports. This acceptance represents the authorization decision to use Galactic by the CC.

6) Monitoring the effectiveness of the security controls: The framework collects the defined security metrics as per the assessment plan of each tenant and generates status reports to the intended cloud stakeholders. A report shows the metrics status and trends, as shown in Figure 7.

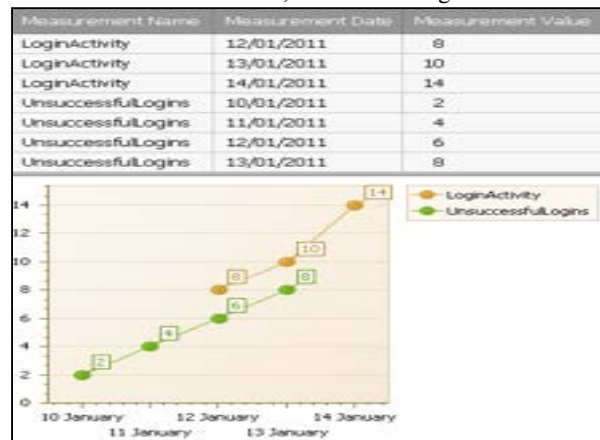


Figure 7: Sample of Swinburne security status report

VII. DISCUSSION

The procedure we went through in the example above should be applied not only for published services but also on the cloud platform services themselves. In this case the CP uses our framework to manage the platform security from a consumer perspective. We have done this for the Galactic exemplar used above.

Our approach provides a security management process; a set of standards-based models for describing platforms, platform services, and services; the security needs of different stakeholders; known threats, risks and mitigations for a cloud deployment; and a tool supporting security plan development and partial automation of a derived security plan. Our approach is comprehensive, supporting all stakeholder perspectives, and collaborative, allowing different stakeholders to develop a mutually-satisfying security model. It addresses the multi-tenancy nature of shared cloud-hosted services when tenants have different security requirements and different SMPs. This is achieved by maintaining and managing multiple security profiles with multiple security controls on the same service. Such controls are delivered by security vendors. This also enables managing traceability between controls, the identified risks and identifies what are the risks still not mitigated.

The SMP of a cloud service has two possible scenarios: Either to let each tenant go through the whole SMP as if he is the only user of the service (tenant-based SMP) or to accumulate all tenants security requirements on a given service and maintain the SMP on the service level (service-based SMP). The later scenario is more straight forward because cloud stakeholders collaborate together to secure the cloud platform and their services with one set of security requirements. The former scenario gives the CCs more control in securing their cloud hosted asset but it has the following problems: (i) the current multi-tenancy feature delivered by the cloud services enables tenants to customize service functionality but it does not enable tenants to customize service security capabilities; (ii) the underlying cloud platform infrastructure, such as OS, does not support for multi-tenancy, so we cannot install multiple anti-viruses or anti-malware systems on the same OS while being able to configure each one to monitor specific memory process for a certain user. One solution may be to use a VM for each tenant as in [7]. This work around may not be applicable if the service is not designed for individual instances usage or if the cloud platform does not support VM technology.

Whenever the CCs are not interested in following the security standards or require a light-weight version of our approach, they can leave out as many steps as they want including security controls implementation, security assessment and service authorization steps. The mandatory steps are service categorization and controls selection. Another variation of our framework is to enable CPs to deliver predefined security versions for the service. CCs can select the suitable version based on their security needs.

We are exploring the cloud security engineering and security controls development processes to develop more flexible services to fit with cloud requirements. Our framework also needs further extension of the automation of the security controls implementation phase. This requires being able to transform from our security plan template configurations into specific security controls configuration. We also plan to derive such configuration parameters' values from the current environment security status.

VIII. SUMMARY

In this paper we introduced a collaboration-based security management framework for the cloud computing model. The framework introduces an alignment of the NIST-FISMA standard to fit with the cloud computing model. We utilize the existing security automation efforts such as CPE, CWE, CVE and CAPEC to facilitate the cloud services Security Management Process (SMP). We have validated our framework by using it to model and secure a multi-tenant SaaS application with two different tenants. The framework can be used by cloud providers to manage their cloud platforms, by cloud consumers to manage their cloud-hosted assets, and as a security-as-a-service to help cloud consumers in outsourcing their internal SMP to the cloud.

ACKNOWLEDGEMENTS

Funding for parts of this research by the FRST SPPI project and Swinburne University of Technology is gratefully acknowledged.

REFERENCES

- [1] International Data Corporate (IDC), "Ranking of issues of Cloud Computing model," 2010. <<http://blogs.idc.com/ie/?p=730>> Accessed Dec 2010.
- [2] M. Almsory, J. Grundy, I. Mueller, "An analysis of the cloud computing security problem," In the proc. of the 2010 Asia Pacific Cloud Workshop, Colocated with APSEC2010, Australia, 2010.
- [3] Cloud Security Alliance Group, "CSA-GRC Stack," <www.cloudsecurityalliance.org/grcstack.html>, Accessed Dec'10
- [4] Unofficial FedRAMP Community Collaboration, <<http://www.fedramp.net/tiki-index.php>>, Accessed in Aug 2010.
- [5] NIST, "Standards for Security Categorization of Federal Information and Information Systems. FIPS-199", <csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199.pdf>, Accessed Dec 2010.
- [6] International Organization for Standardization (ISO), "ISO/IEC 27000 - Information technology - Security techniques - Information security management systems - Overview and vocabulary," ISO/IEC 27001:2005(E), 2009, <http://webstore.iec.ch/preview/info_isoiec27000%7Bed1.0%7Den.pdf>, Accessed in July 2010.
- [7] M. Menzel, R. Warschofsky, et al, "The Service Security Lab: A Model-Driven Platform to Compose and Explore Service Security in the Cloud," 6th World Congress, SERVICES2010, pp.115-122.
- [8] M. Menzel and C. Meinel, "SecureSOA Modelling Security Requirements for Service-Oriented Architectures," IEEE International Conference on Services Computing, 2010.
- [9] S. Bertram, M. Boniface, et al., "On-Demand Dynamic Security for Risk-Based Secure Collaboration in Clouds," *IEEE 3rd International Conference in Cloud Computing*, pp. 518-525, 2010.
- [10] P. Saripalli and B. Walters, "QUIRC: A Quantitative Impact and Risk Assessment Framework for Cloud Security," *IEEE 3rd International Conference on Cloud Computing*, 2010, pp. 280-288.
- [11] Z. Xuan, N. Wuwong, et al., "Information Security Risk Management Framework for the Cloud Computing Environments," in *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010, pp. 1328-1334.
- [12] NIST, "Risk Management Guide for Information Technology Systems," 2002, <<http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>>, Accessed in June 2010.
- [13] P. Marek and J. Paulina, "The OCTAVE methodology as a risk analysis tool for business resources," presented at the International Multiconference Computer Science and IT, Hong Kong, 2006.
- [14] R. Fredriksen, M. Kristiansen, et al, "The CORAS Framework for a Model-Based Risk Management Process," in *Computer Safety, Reliability and Security*, vol. 2434, Springer, 2002, pp. 39-53.
- [15] C. Basile, A. Liroy, et al, "POSITIF: A Policy-Based Security Management System," in *8th IEEE International Workshop Policies for Distributed Systems and Networks*, 2007, pp. 280-280, Italy.
- [16] H. Xu, X. Xia, et al "Towards Automation for Pervasive Network Security Management Using an Integration of Ontology-Based and Policy-Based Approach," *3rd International Conference Innovative Computing Information and Control*, 2008, pp. 87-87, Dalian.
- [17] J. Albuquerque, H. Krumm and P. de Geus, "Model-based management of security services in complex network environments," in *IEEE Network Operations and Management Symposium*, 2008, pp. 1031-1036, Salvador.
- [18] S. de Chaves, C. Westphall and F. Lamin, "SLA Perspective in Security Management for Cloud Computing," in *Sixth International Conference Networking and Services*, 2010, pp. 212-217, Mexico.
- [19] P. Patel, A. Ranabahu and A. Sheth, "Service Level Agreement in Cloud Computing," Conference on Object Oriented Programming Systems Languages and Applications, Orlando, Florida, 2009, USA.
- [20] NIST, "The Federal Information Security Management Act (FISMA)," U.S. Government Printing 2002, <<http://csrc.nist.gov/drivers/documents/FISMA-final.pdf>>, Accessed on August 2010.
- [21] Mitre Corporation. (2010), *Making Security Measurable*, <<http://measurablesecurity.mitre.org/>>, Accessed on Jan 2011

5.4 DCTracVis: a system retrieving and visualizing traceability links between source code and documentation

Chen, X., Hosking, J.G., Grundy, J.C., Amor, R., DCTracVis: a system retrieving and visualizing traceability links between source code and documentation, *Automated Software Engineering*, vol 25, no 4, 2018, Springer, pp. 703–741

DOI: [10.1007/s10515-018-0243-8](https://doi.org/10.1007/s10515-018-0243-8)

Abstract: It is well recognized that traceability links between software artifacts provide crucial support in comprehension, efficient development, and effective management of a software system. However, automated traceability systems to date have been faced with two major open research challenges: how to extract traceability links with both high precision and high recall, and how to efficiently visualize links for complex systems because of scalability and visual clutter issues. To overcome the two challenges, we designed and developed a traceability system, DCTracVis. This system employs an approach that combines three supporting techniques, regular expressions, key phrases, and clustering, with information retrieval (IR) models to improve the performance of automated traceability recovery between documents and source code. This combination approach takes advantage of the strengths of the three techniques to ameliorate limitations of IR models. Our experimental results show that our approach improves the performance of IR models, increases the precision of retrieved links, and recovers more correct links than IR alone. After having retrieved high-quality traceability links, DCTracVis then utilizes a new approach that combines treemap and hierarchical tree techniques to reduce visual clutter and to allow the visualization of the global structure of traces and a detailed overview of each trace, while still being highly scalable and interactive. Usability evaluation results show that our approach can effectively and efficiently help software developers comprehend, browse, and maintain large numbers of links.

My contribution: Developed initial ideas for this research, co-designed approach, co-supervised PhD student, co-authored significant parts of paper, co-lead investigator for funding for this project from FRST

DCTracVis: a system retrieving and visualizing traceability links between source code and documentation

Xiaofan Chen • John Hosking • John Grundy • Robert Amor

Abstract It is well recognized that traceability links between software artifacts provide crucial support in comprehension, efficient development, and effective management of a software system. However, automated traceability systems to date have been faced with two major open research challenges: how to extract traceability links with both high precision and high recall, and how to efficiently visualize links for complex systems because of scalability and visual clutter issues. To overcome the two challenges, we designed and developed a traceability system, DCTracVis. This system employs an approach that combines three supporting techniques, Regular Expressions, Key Phrases, and Clustering, with Information Retrieval (IR) models to improve the performance of automated traceability recovery between documents and source code. This combination approach takes advantage of the strengths of the three techniques to ameliorate limitations of IR models. Our experimental results show that our approach improves the performance of IR models, increases the precision of retrieved links, and recovers more correct links than IR alone. After having retrieved high-quality traceability links, DCTracVis then utilizes a new approach that combines treemap and hierarchical tree techniques to reduce visual clutter and to allow the visualization of the global structure of traces and a detailed overview of each trace, while still being highly scalable and interactive. Usability evaluation results show that our approach can effectively and efficiently help software developers comprehend, browse, and maintain large numbers of links.

This research is financially supported by the Foundation for Research, Science and Technology, MBIE Software Process and Product Improvement project, and University of Auckland

Xiaofan Chen
School of Computer Science and Engineering, Nanjing University of Science and Technology, China
e-mail: xiaofanchen@hotmail.com

John Hosking
Dean of Science, University of Auckland, New Zealand
e-mail: j.hosking@auckland.ac.nz

John Grundy
Senior Deputy Dean for the Faculty of Information Technology, Monash University, Australia
e-mail: john.grundy@monash.edu

Robert Amor
Department of Computer Science, University of Auckland, New Zealand
e-mail: trebor@cs.auckland.ac.nz

Keywords Software traceability · Traceability recovery · Traceability visualization

1 Introduction

It is well recognized that traceability links between artifacts play a critical role in program comprehension, maintenance, requirements tracing, impact analysis, reuse and management of a software system (Antoniol et al., 2000; Gotel and Finkelstein, 1994; Settini et al., 2004; Watkins and Neal, 1994). Unfortunately, it is also painstaking, error-prone, complex and time-consuming work to manually retrieve and maintain traceability links between artifacts. These efforts can be significantly reduced by applying traceability recovery approaches to automatically obtain high quality relationships and links between elements in one artifact and elements in another (Penta et al., 2002; Settini et al., 2004; Spanoudakis and Zisman, 2005), and adopting traceability visualization techniques to represent these retrieved links in a natural and intuitive way (Asuncion et al., 2007; Roman and Cox, 1992). High quality links represent a link set containing as many as possible correct links and as few as possible incorrect links. Moreover, high quality links connect elements of different artifacts on a fine-grained level of detail e.g. part of a design document description and its related source code elements.

Most automatic traceability recovery approaches (Antoniol et al., 2002; Cleland-Huang et al., 2005; Hayes et al., 2003; Lucia et al., 2007; Marcus and Maletic, 2003; Wang et al., 2009) use Information Retrieval (IR) models to extract links between artifacts. IR is an area that studies the problem of finding relevant information in text collections based on user queries (Hayes et al., 2003; Spanoudakis and Zisman, 2005). In other words, IR models determine how relevant a piece of text is to a query that represents a user's interest by computing a similarity value according to the frequency and distribution of keywords or terms in textual format document collections (Hayes et al., 2003; Spanoudakis and Zisman, 2005). The precision of the extracted traceability links heavily depends on a threshold that decides which links can be recovered. There are two ways to determine the threshold (Lucia et al., 2007; Marcus and Maletic, 2003). One way is to determine a threshold for the similarity value that identifies which documents are linked. Only links that have a similarity value greater than the threshold will be captured. Another way is to impose a threshold on the number of retrieved links, regardless of the actual similarity value. This means that the user can decide to retrieve the top ranked links among those that have a similarity value greater than the threshold. However, the lower the threshold is the greater the number of incorrect links that are retrieved. Conversely, the higher the threshold is the lesser the number of correct links that are retrieved. This means that many potentially useful and important links are missed at high thresholds. Similarly, many incorrect or unuseful links are extracted at low thresholds and may confuse developers. Furthermore, the same threshold may or may not be best suited for different systems.

While traceability links between artifacts are captured by a traceability recovery technique, a remaining key issue is how to represent these retrieved links to assist software engineers to effectively and efficiently understand, browse, and maintain them. Adopting software visualization techniques (e.g. tree-based, graph-based, or 3D-based approaches) is a common way to display retrieved links (Asuncion et al., 2007; Roman and Cox, 1992). However, displaying a great many traceability links effectively and efficiently is a big challenge, because a software system with large numbers of artifacts, and thus very large numbers of traceability links between artifacts, quickly gives rise to scalability and visual clutter issues (Cornelissen et al., 2007; Holten, 2006; Merten et al.,

2011). Moreover, the efficient visualization of both the structure of the traced system and the enormous number of links between artifacts is a far from trivial problem (Cornelissen et al., 2007; Marcus et al., 2005). Many traceability visualization techniques (Cleland-Huang and Habrat, 2007; Cornelissen et al., 2007; Merten et al., 2011; van Ravensteijn, 2011; Zhou et al., 2008) have been designed and developed to represent traceability links. To date, however, no traceability visualization techniques can visualize a great many traceability links effectively and efficiently without scalability and visual clutter issues.

In order to remedy these issues, we designed and developed a traceability system, called DCTracVis, to capture and visualize traceability links between artifacts efficiently and effectively. Our traceability system employs a new traceability recovery approach that can automatically recover high quality links between artifacts in the traced system at all cut points, and a new traceability visualization technique to visualize retrieved links in a natural and intuitive way.

Our traceability recovery approach combines Information Retrieval (IR) models with three supporting techniques: Regular Expressions (RE), Key Phrases (KP), and Clustering. These particular techniques have quite different strengths and weaknesses and recover different sets of links due to their vastly different retrieval approaches. Our recovery approach attempts to take advantage of the different strengths of the three enhancement techniques to increase precision at low levels of threshold and recall at high levels of threshold of links recovered by IR. We have conducted a detailed experiment to evaluate our recovery approach by applying six IR models to four case studies varying in size and context. Analysis of the experimental results illustrates that a combination of the three enhancement techniques can be used effectively to improve precision and recall of links retrieved by IR at all thresholds.

Our traceability visualization technique combines enclosure and node-link representations to reduce visual clutter and to allow the visualization of the global structure of traces and a detailed overview of each trace, while still being highly scalable and interactive. We have adopted two visualization techniques to achieve these goals: treemap and hierarchical tree. A treemap view displays a tree structure by means of enclosure and provides an overview of inter-relationships between artifacts. In order to reduce visual clutter, we employ colours to represent the relationship status of each node in the treemap, instead of directly drawing edges between related nodes on top of the treemap. We use two hierarchical trees that can be expanded and contracted to visualize links. One hierarchical tree visualization is used to illustrate detailed link information about each trace. The other is used to display the whole project under trace and traceability links in it to communicate the hierarchical structure of the project. Our traceability system also includes navigation, search, and filter functions to help engineers locate particular nodes and filter out uninteresting links. We have conducted a usability study to assess the usefulness of our traceability system for large traceability visualization problems. The results of this evaluation show that our system is both easy to use and can effectively and efficiently help software developers recover traceability links and comprehend, browse, and maintain large numbers of links.

Our particular focus in this research is on traceability between classes in source code and sections in documents that are written in natural language and are produced during the software development process, e.g. requirements, design documents, tutorials, developer or user guides, and emails. The objective of our research is to provide software

engineers with an effective visualization environment enabling them to retrieve, create, browse, edit, and maintain traceability links between artifacts effectively and efficiently. With this environment, engineers can trace relationships between various documents and source code, automatically recover traceability links at low cost and high accuracy, easily create and change links as well as conveniently browse and maintain links. In terms of size, we are interested in systems with potentially several hundreds to even thousands of classes, dozens if not hundreds of documents, and many tens of thousands to hundreds of thousands of traceability links between classes and document elements.

The rest of this paper is organized as follows. Related work is discussed in Section 2. Section 3 describes approaches for retrieving traceability links and visualizing them. A description of our traceability system and its implementation are described in Sections 4 and 5. Section 6 presents the evaluation results, followed by the analysis of these results in Section 7. Finally, we draw conclusions in Section 8.

2 Related work

Due to the importance of software traceability, extensive effort in the software engineering research community has been put into improving the precision and recall of recovered traceability links between artifacts through various traceability recovery techniques and the visualization of retrieved links in a natural and intuitive way through traceability visualization techniques.

2.1 Traceability link recovery

The most studied and often used techniques in automated traceability link recovery to date are Information Retrieval (IR) models (Antoniol et al., 2002; Cleland-Huang et al., 2005; Marcus and Maletic, 2003; Settini et al., 2004; Wang et al., 2009). Early IR systems were Boolean models which use a complex combination of Boolean ANDs, ORs, and NOTs to specify users' needs (Hayes et al., 2003; Singhal, 2001). However, Boolean models are not very effective as they do not support ranked retrieval. Therefore, current IR models rank documents by their estimation of the relevance of a document for a query. Most of them assign a similarity value to every document and rank documents by this value. (Hayes et al., 2003; Singhal, 2001)

Antoniol et al. (2002) applied two different IR models, Probabilistic Model (PM) and Vector Space Model (VSM), to extract links between code and documentation. Their results show that IR provides a practical solution for automated traceability recovery, and the two IR models have similar performance when terms in artifacts perform a preliminary morphological stemming. However, PM and VSM produce links at low levels of precision and reasonable levels of recall. Marcus and Maletic (2003) introduced Latent Semantic Indexing (LSI), an extension of the VSM, to recover links between documentation and source code. Their results show that although LSI achieves very good performance without the need for stemming, as required for PM and VSM, it suffers from the problem of low precision and high recall at low levels of thresholds or high precision and low recall at high levels of thresholds.

In order to improve the performance of IR models, many strategies have been developed. A traceability recovery tool based on PM was developed to explore how the

retrieval performance can be improved by modeling programmer behavior (Antoniol et al., 2000). Their results show that improvement in recall is achieved (Antoniol et al., 2000; Lucia et al., 2007). Cleland-Huang et al. (2005) proposed an approach to improve the performance of dynamic requirements traceability by incorporating three different strategies into PM, namely hierarchical modeling, logical clustering of artifacts, and semi-automated pruning of the probabilistic network. Their results indicate that the three strategies effectively improve trace retrieval performance. Nevertheless, the three strategies have varying abilities to enhance link retrieval and are unable to work in all cases.

Settimi et al. (2004) investigated the effectiveness of VSM and VSM with a general thesaurus for generating links between requirements, code, and UML models. The comparison results show that precision and recall are not improved by the use of the general thesaurus. Hayes et al. (2003) used VSM but with a context-specific thesaurus that is established based on technical terms in requirement documents to recover links between requirements. The results show that improvements in recall and sometimes in precision are achieved. Nishikawa et al. (2015) proposed the Connecting Links Method (CLM) to recover transitive traceability links. This approach first employs VSM to extract links between X and Z and between Y and Z, then connects these links to recover links between X and Y. Their results show that CLM is more effective than VSM alone as CLM can recover links in cases where VSM does not.

Wang et al. (2009) presented four enhanced strategies to improve LSI, namely, source code clustering, identifier classifying, similarity thesaurus, and hierarchical structure enhancement. Their comparison results indicate that this approach has higher precision than LSI and PM, but has lower recall. To improve IR-based traceability recovery, namely from the VSM, LSI and Jensen-Shannon models (JS), Lucia et al. (2013) applied a smoothing filter to remove “noise” from the textual corpus of artifacts. Their study indicates that a smoothing filter can remove “noise” that simple stop word filters cannot remove and can significantly improve the performances of traceability recovery. Falessi et al. (2017) proposed a new approach called ENRL which employs Machine Learning classifiers to estimate the number of remaining positive links in a ranked list of candidate traceability links produced by an NLP-based recovery approach (e.g. VSM, TF-IDF, or Latent Semantic Analysis etc.). Their results indicate that ENRL can provide accurate estimates of the number of remaining positive links but depends on the choice of the NLP technique. Kuang et al. (2017) combined IR techniques with closeness analysis to improve IR-based traceability recovery. The closeness analysis is to quantify the degree of interaction based on direct and indirect code dependencies among classes. Their results show that this approach outperforms IR-based approaches (e.g. VSM, JS, and LSI etc.).

Although various strategies have been applied to enhance the performance of IR techniques, no approaches can significantly decrease incorrect (fault) links at low levels of threshold and significantly increase correct (true) links at high levels of threshold (Antoniol et al., 2002; Cleland-Huang et al., 2005; Marcus and Maletic, 2003; Settimi et al., 2004; Wang et al., 2009). Our hypothesis is that augmenting IR with several complementary techniques - Regular Expressions, Key Phrases and Clustering - will significantly enhance both precision and recall.

2.2 Traceability link visualization

Software engineers traditionally store or represent traceability links in tabular formats using a spread-sheet, matrix, cross-references, or a database. Matrix and cross-reference techniques are very common traditional methods of representing traceability links. A traceability matrix is easy to understand and provides a quick overview of relations between two artifacts if the set of artifacts is small (van Ravensteijn, 2011; Li and Maalej, 2012). However, the matrix misses the inherent hierarchy and becomes unreadable when the set of artifacts becomes large (Voytek and Nunez, 2011). The cross-reference pattern is also easy to understand but cannot provide the overall structure of traces (van Ravensteijn, 2011). It is difficult to identify individual traceability links as they are lost in this table structure. The approach, therefore, does not scale to large numbers of classes and documents.

More recently, research has focused on displaying links in a graph or tree due to the convenience and ease of browsing and understanding the links (Li and Maalej, 2012). Graph-based visualization techniques represent artifacts as nodes and traceability links between artifacts as edges to form a graph. Graphs can show the overall overview of relationships between artifacts and allows one to easily browse links.

ADAMS (Lucia et al., 2004) supports specifying links between pairs of artifacts. Traceability links are organized in a graph where nodes are represented by the artifacts and edges are the traceability links. After users select a source artifact, the graph is built starting from a source artifact by finding all the dependencies of a specific type that involve the source artifact either as source or target artifact (ADAMS, 2009). Within the graph, users can identify traceability paths, i.e. sets of artifacts connected by traceability links. This graph performs very well in displaying all links of a selected source artifact. However, it fails to support the display of multiple artifacts' links. Cleland-Huang and Habrat (2007) proposed a hierarchical graphical structure to visualize links, in which leaf nodes are represented by requirements while titles and other hierarchical information are represented as internal nodes. This graph visualization provides a birds-eye-view of the candidate links and their distribution across the set of traceable artifacts. It also allows the user to explore groups of candidate links that naturally occur together in the document's hierarchy (Cleland-Huang and Habrat, 2007). Unfortunately, this visualization becomes very large as the data set gets bigger. Moreover, it uses the display space inefficiently. Zhou et al. (2008) developed ENVISION, adopting a hyperbolic tree view with the enhancement of a "focus+context" approach to facilitate software traceability understanding. The results of their empirical study show that this view allows users to maintain a global view of links as well as being able to dive deep into an interesting traceability path. However, this view is also not space-efficient. Kamalabalan et al. (2015) developed a tool that visualizes relationships as a graph with nodes and edges. This tool uses the Neo4j Graph Database for modeling relationships. It allows users to overview the overall structure and to get more in-depth details using cluster views of filtered artefacts or relationships. Though the cluster views reduce visual clutter, the overview graph is getting larger and more complex when displaying a larger volume of data. Nakagawa et al. (2017) visualized the traceability links between specifications and test case descriptions in macro and micro views. In the macro view, nodes correspond to specifications and edges represent relatively high similarities between two specifications. The size of a node represents the number of assigned test case descriptions. The micro

view contains a specification list; each specification item is linked to a page that lists its related test case descriptions. However, the views have visual clutter issues.

TBreq (LDRA, 2012), a commercial application, provides end-to-end traceability from requirements to design, code, and test. It lists artifacts horizontally and draws linear edges between related items of artifacts. It cannot provide the hierarchical structure and can quickly produce severe visual clutter for a system with medium to large numbers of artifacts. TraceVis (van Ravensteijn, 2011; van Amstel et Al., 2012), visualizes a dynamic list of hierarchies and adjacency relations. It uses icicle plots and hierarchical edge bundling (Holten, 2006) techniques to support the hierarchical structure and to reduce visual clutter. Icicle plots are used to represent hierarchies vertically. Adjacent relations are represented by drawing edges between related items. Edges are displayed using splines and are grouped using hierarchical edge bundling. TraceVis supports an overview of, as well as a detailed insight into, inter-related, hierarchically organized data. However, it uses space inefficiently and can result in visual clutter if the dataset is large or lateral relations are visualized (van Ravensteijn, 2011; van Amstel et Al., 2012). Rocco et al. (2013) used TraceVis to visualize the dependencies between artefacts and their related metamodel, and to help understanding how and where changes affect the system.

Merten et al. (2011) utilized sunburst and netmap techniques to display traceability links between requirements knowledge elements. The sunburst visualizes the hierarchical structure of the project under trace. Nodes are arranged in a radial layout and are displayed on adjacent rings representing the tree structure. The netmap aims to represent links between requirements. The nodes in a netmap are in a circle and are segments of exactly one ring in the sunburst. Traceability links are drawn by using linear edges in the inner circle. Although the two techniques can visualize the overall hierarchical structure and can easily browse links, the graph can become very large, leading to visual clutter when dealing with a large number of traceability links. EXTRAVIS, developed by Cornelissen et al. (2007) employs a hierarchical edge bundling technique (Holten, 2006) that groups edges based on the structure of a hierarchy to reduce the visual clutter. Using a circular bundle view shows the structure of the system under trace and represents execution traces. The hierarchies are shown by using an icicle plot based on a mirrored layout. A global overview of traces is provided by a massive sequence view. However, when considering a large number of traces, it becomes difficult to discern the various colors and to prevent bundles overlapping. Multi-VisioTrace (Rodrigues et al., 2016) supports multiple visualization techniques: matrix view, tree view, sunburst and graph view. It allows users to choose the most appropriate to their tasks. This tool still leads to visual clutter when displaying a large volume of data.

In addition to traditional approaches and the various graph representations similar to those reviewed above, there are several other approaches that have been used to visualize traceability links. Poirot (Cleland-Huang and Habrat, 2007; Cleland-Huang et al., 2007) displays trace results in a textual format. It uses confidence levels, user feedback checkboxes, and tabs separating likely and unlikely links to assist the analyst in evaluating candidate links. However, it cannot visualize overall structure. TraceViz (Marcus et al., 2005) employs a map consisting of coloured and labeled squares to display traceability links for a specific source or target artifact. It allows users to clearly visualize all links of a selected source artifact or a chosen target artifact. Unfortunately, it

is unable to display links for multiple artifacts at the same time. LeanArt (Grechanik et al., 2007) utilizes an intuitive point-and-click graphical interface to enable users to navigate to program entities linked to elements of UCDs by selecting these elements, and to navigate to elements of UCDs by selecting program entities to which these elements are linked. The characteristic of LeanArt is to select a source, and it then displays targets linked to this source. It also fails to present all links at the same time. A 3D approach (Pilgrim et al., 2008) is introduced to enhance traceability visualization between UML diagrams. Artifacts are projected on layered planes. Traces between different levels of abstraction are visualized by using edges between planes. Although presenting more content at once and grouping related information together, the 3D approach adds more complexity to the graph, and still leads to visual clutter when the data set becomes large.

To varying degrees, none of the traceability visualization techniques developed so far can visualize a great many traceability links effectively and efficiently without scalability and visual clutter issues. Users of such link visualizations not only need scalable, effective representations, but must also be able to navigate complex software systems and their documentation to help them recover, browse, and maintain inter-relationships between artifacts in a natural and intuitive way (Cornelissen et al., 2007; Holten, 2006; Marcus et al., 2005; Merten et al., 2011).

These issues motivated us to develop a visualization technique to enable engineers to recover, browse, modify, and maintain links effectively and efficiently. The discussion above on visualization techniques showed that combining different visualization approaches can display elements efficiently. For example, combining node-link representations and enclosure offers a trade-off between an intuitive display and efficient space usage for visualizing large numbers of artifacts in a system (Graham and Kennedy, 2010; Holten, 2006; Shneiderman, 1992); node-link representations (e.g. the hierarchical tree) that communicate structure readily, and the enclosure layout (e.g. the treemap) which is very effective for displaying large numbers of elements. Similar to our approach of combining several traceability recovery techniques to mitigate each other's weaknesses, our visualization approach combines several visualization techniques to provide more effective and efficient link visualization.

3 Our approach

We have developed a traceability system, called DCTracVis, to support software engineers to recover, browse, understand, and maintain links efficiently and effectively. Our traceability system employs a new traceability recovery technique to retrieve traceability links between artifacts and a new traceability visualization technique to display these retrieved links. Section 3.1 describes our traceability recovery technique. Section 3.2 presents our traceability visualization technique. Other functions provided in the DCTracVis are described in Section 3.3

3.1 Traceability link recovery

In order to improve the accuracy of retrieved traceability links to a reasonably high level at all levels of threshold, we have been exploring a new approach combining Regular Expressions (RE), Key Phrases (KP), and Clustering techniques with IR models to

recover links between sections in documents and class entities. In our previous work (Chen and Grundy, 2011), we focused on generating traceability links by combining the Vector Space Model (VSM) with RE, KP and Clustering techniques. We showed that this combination approach provided better performance than VSM alone. In this paper, we broaden our previous work to investigate whether IR models can recover traceability links with high precision and recall at any level of threshold by combining a range of different IR models with these three enhancement techniques.

Our basic retrieval technique uses different IR models to recover links between class entities and sections. Table 1 shows the six IR models used in this paper: VSM (Chen and Grundy, 2011), TF_IDF, PL2, BM25, DLH, and IFB2. (For more details on the other five IR models, we refer the reader to work on the Terrier tool (Terrier IR platform, 2010).)

Table 1 A Description of IR Retrieval Models

IR models	Description
VSM	Vector space model by constructing vector representations for documents
TF_IDF	The $tf*idf$ weighting function, where tf is given by Robertson's tf and idf is given by the standard Sparck Jones' idf .
PL2	Poisson estimation for randomness, Laplace succession for first normalization, and Normalization 2 for term frequency normalization.
BM25	The BM25 probabilistic model ranks documents based on query terms appearing in each document, regardless of the interrelationship between query terms within a document.
DLH	The DLH hyper-geometric Divergence From Randomness (DFR) model, parameter-free weighting model.
IFB2	Inverse Term Frequency model for randomness, the ratio of two Bernoulli's processes for first normalization, and Normalization 2 for term frequency normalization.

As many papers have extensively discussed these IR models (Antoniol et al., 2002; Cleland-Huang et al., 2005; Hayes et al., 2003; Lucia et al., 2007; Marcus and Maletic; 2003; Wang et al., 2009), we only briefly describe how IR queries are built. IR queries, to find text relevant to a class name, include class names and their constituent words if a class name is formed by compound words. A class name (or identifier) composed of two or more words is split into separate words. An IR query string is established by using the OR operator to combine the name and the separate words. For example, PrinterName is split into the words printer and name, then the query string is “PrinterName OR printer name OR printer OR name”. The query is case-insensitive. IR extracts from a collection a subset of sections that are deemed relevant to a given query and assigns a similarity score ($0 \leq \text{similarity score} \leq 1$) to each retrieved section based on frequency and distribution of key words in the query. This can result in some accurate links having a very low similarity score (Antoniol et al., 2002; Cleland-Huang et al., 2005; Hayes et al., 2003; Lucia et al., 2007; Marcus and Maletic; 2003; Wang et al., 2009). The lower the threshold that is used, the more possible links are retrieved but the more fault links are captured as well. In other words, at a high threshold, IR captures few links with few positive links.

In order for us to augment the number of retrieved links at high threshold levels, the RE technique is used to find all of the occurrences of class names in documents. It uses two regular expressions (using the class “Control” for the example): $(.*)^{(a-zA-Z0-9)}<C-?o-?n-?t-?r-?o-?l>^{(a-zA-Z0-9)}(.*)$ for matching class names in documents; $(.*)^{(a-zA-Z0-9)}<\text{each part of package name}>^{(a-zA-Z0-9)}(.*)$ for matching each part of package names. The two REs are automatically built to tailor the different data sets. We use the KP technique to extract key words (or key phrases) from comments of code to

provide a brief summary of each class's description comment and add these to IR queries to augment our IR model link recovery. We utilize the Clustering technique to reduce fault links by using the inherent hierarchical structure in documents. We modify the K-mean clustering algorithm (MacQueen, 1967) to discard links that are not assigned in clusters. For a detailed discussion of the three enhancement techniques, please refer to Chen and Grundy (2011).

Our approach is intended to overcome the limitations of IR techniques by taking advantage of the strengths of RE, KP, and Clustering. Combining RE with IR models allows extraction of more correct links at high thresholds. As long as class names are retrieved correctly and refined regular expressions are built, RE can retrieve all possible links that are related to these class names and return few incorrect links as well. Adding KP enables IR to generate all potential links by extending the IR queries to include key phrases from comments in the source code. If source code is well documented, KP can extract key phrases from comments closely related to classes. The majority of incorrect links at low thresholds are discarded by adopting Clustering, which takes advantage of the inherent hierarchical structure of documents to cluster links retrieved by IR models, RE, and KP. Therefore, our combination approach increases the number of correct links at high thresholds and reduces the number of incorrect links at any threshold.

3.2 Traceability link visualization

In order to provide efficient traceability visualization, we have explored an approach of combining enclosure and node-link representations to display the overall structure of traceability links and provide a detailed overview of each link while still being highly scalable and interactive. We utilize two visualization techniques to achieve these goals: treemap and hierarchical tree. The treemap view is adopted to display the structure of the system under trace and the overall overview of links. We utilize colours to differentiate the relationship status of each node in the treemap instead of drawing edges directly over the treemap. The latter approach quickly leads to visual clutter. We adopt two hierarchical trees that can be expanded and contracted to visualize links. A whole hierarchical tree (the whole HT) is used to display the whole system and links in it to communicate the hierarchical structure of the system. When an item is selected in the treemap view or the whole HT, a detail hierarchical tree (the detail HT) is built to provide the detailed dependency information of the selected item. The detailed HT is treated as a supplement to the treemap and the whole HT. Any change to links made in the treemap is reflected in the two hierarchical trees, and vice versa. The previous work presented in Chen et al. (2012) focused on visualizing links using Treemaps and Hierarchical trees. In this paper, we extend our previous prototype to include three more functions: Navigator, Search and Filter. Navigator and Search functions are provided to assist users to find a specific node. For the filter function, four methods are used to filter out traceability links: IR model, combined traceability recovery approach, the similarity score level, and the number of links level. The following sections describe the visualization techniques and how we support editing of links.

3.2.1 Treemap view

The treemap technique adopts a space-filling layout technique to represent a tree structure by means of enclosure, which places child nodes within the boundaries of their parent nodes and encloses each group of siblings by a margin (Shneiderman, 1992). This layout makes it an ideal technique for displaying a large tree and using display space effectively (Graham and Kennedy, 2010; Holten, 2006; Shneiderman, 1992). Although the treemap technique cannot communicate the hierarchical structure very well, it can convey the high-level, global structure of a system under trace. It is also effective in helping to answer questions such as what artifacts the system has, how many items each artifact has, which artifact contains the most numbers of items, and how artifacts are organized.

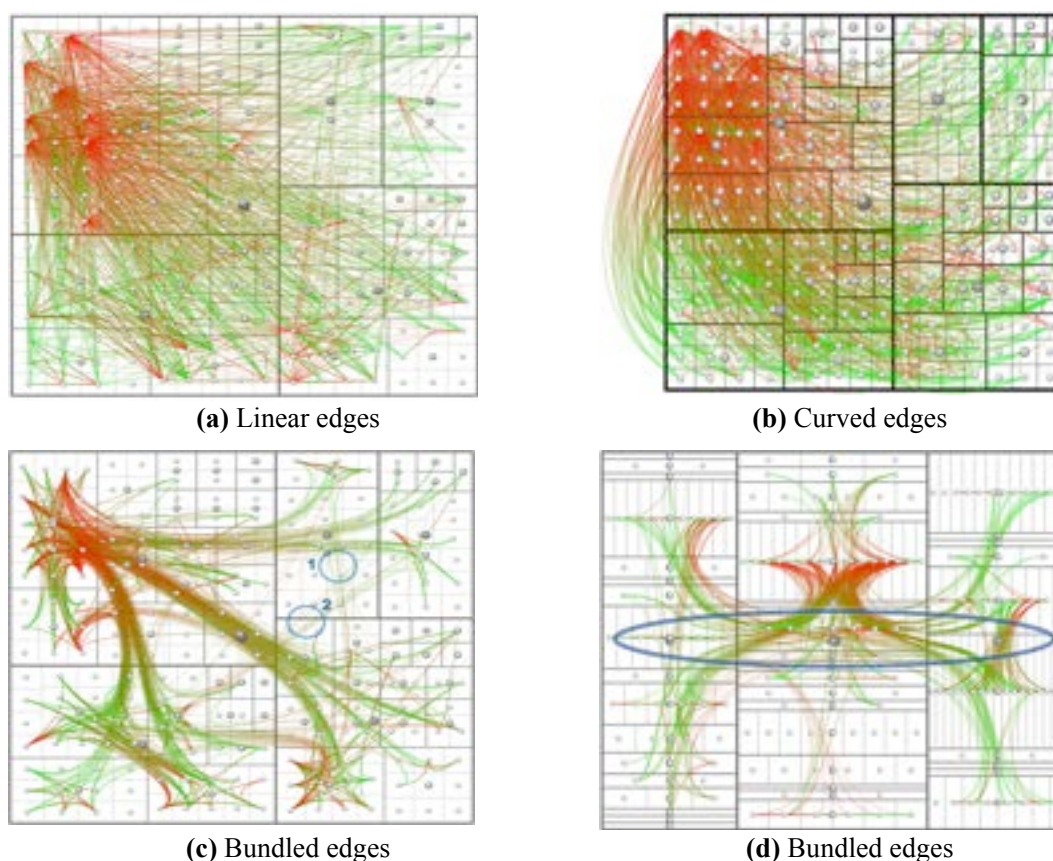








Figure 1 Displaying Traceability Links between Nodes Using (a) Straight/linear edges; (b) Curved link edges; (c) and (d) Edges grouped by Hierarchical Edge Bundling. (Holten, 2006)

In order to display traceability links between artifacts in a treemap, the straightforward way is to add relationships between related nodes as edges over the treemap, as in (Holten, 2006) (see Figure 1). Figure 1a shows straight and linear edges between related nodes on top of the treemap. Figure 1b uses curved link edges. These two approaches quickly lead to visual clutter if large numbers of edges are displayed. Using a hierarchical edge bundling technique can alleviate this issue. Figure 1c and d group edges based on the structure of a hierarchy (Holten, 2006). However, hierarchical edge bundling can cause bundles to overlap along the collinearity axes (see the encircled region in Figure 1d) if dealing with a large number of collinear nodes in the treemap. All these approaches have difficulty discerning the source and target items of a link if not using other enhancement techniques, e.g. a “focus+context” technique. For example, it is hard to know that edges circled (1 and 2) in Figure 1c are from where to where. Moreover, it is

hard to discern the structure of the system conveyed in the treemap because of the edges drawn on top of the treemap. In addition, it is easy for it to become overcrowded when considering large numbers of links.

In order to ameliorate these issues, we introduce colours to show the relationship status of each node, instead of drawing edges over the treemap. The relationship status of each node describes whether the node has links and how many links it has. We use three colour ranges to show the status of each node (see Table 2 and its application in Figure 4). They are arbitrarily chosen. If a node has fewer than six links, yellow-based colours are used. If the number of links is fewer than 16 but more than 5, gray-based colours are used. Otherwise, we use green-based colours. For each colour range, the shading of the colour indicates intermediate values (lighter implies fewer links, darker more links). Based on the colours of each node without the distraction of the edges on top of the treemap, it is easy to discern the structure of the traced system and an overall overview of the scale of traceability links at the expense of understanding the connectivities.

Table 2 Three Colour Ranges Indicating the Number of Links Each Node has

1. $0 \leq \text{No. of links} < 6$:	Yellow-based		→	
2. $6 \leq \text{No. of links} < 16$:	Gray-based		→	
3. $\text{No. of links} \geq 16$:	Green-based		→	

3.2.2 Hierarchical tree views

The hierarchical tree is an intuitive node-link based representation that uses lines to connect parent and child nodes to depict the relationship between them (Graham & Kennedy, 2010; Holten, 2006). This representation is easy to understand, even to a layperson, and it communicates hierarchical structure very well (Jackson & Wilkerson, 2016; Graham & Kennedy, 2010; Holten, 2006). There are two approaches to visualize traceability links using the hierarchical tree view. The first approach is to draw edges between related children nodes (see Figure 2a). Edges can be grouped using the hierarchical edge bundling technique. However, the approach suffers from overlapping bundles along the collinearity axes (see the encircled region in Figure 2a) and hence visual clutter if dealing with rather large numbers of traceability links (Holten, 2006). The second approach is to directly add traceability links as children of leaf nodes (see Figure 2b). In other words, the original leaf nodes (green circle nodes in Figure 2b) in the hierarchical tree become inner nodes and parents of traceability links (gray rectangle nodes in Figure 2b). For example, if a child node is related to three other nodes, we additionally add the three nodes under the child node. The second approach can ameliorate problems with the first approach.

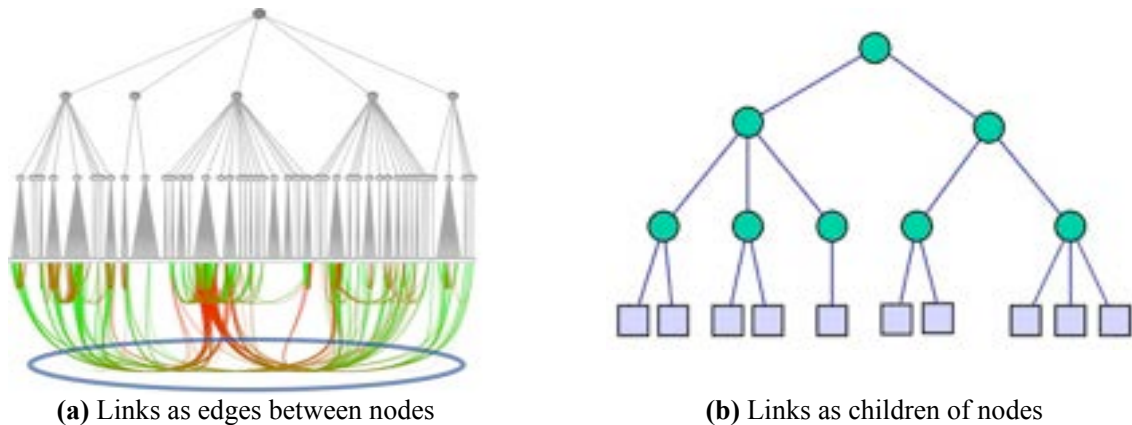


Figure 2 Showing Traceability Links in the Hierarchical Tree Layout: (a) Links as edges between nodes (Holten, 2006), (b) Links as children of nodes

As the connectivity in the treemap is difficult to perceive, we supplement it with a left-to-right hierarchical tree (the whole HT) that can be expanded and contracted to display the whole system under trace (see Figure 5 top). We use this approach to display traceability links as children of artifacts in the system. We also employ the three colour ranges in Table 2 to differentiate the relationship status of each node; whether the node has links and how many it has. However, for nodes with no links (No. of links = 0), they are coloured white to distinguish them from other nodes that have at least one link.

A second left-to-right hierarchical tree layout (“the detail HT”) shows detailed information of a single item once the item is selected in the treemap or the whole HT (see bottom of Figures 4 or 5). This further approach is adopted to display traceability links for the selected item. It illustrates two levels of dependency information. The first level contains artifacts that are related to the selected item. The second level contains other artifacts that are dependent on the artifacts shown in the first level. This view shows not only artifacts related to the item but also dependency information for these artifacts. Moreover, we use red-based colours to show the similarity score levels of links. The darker the colour the higher the similarity score a link has. In addition, providing the hierarchical tree with an ability to expand and contract makes it space-efficient.

Figure 3 shows a sequence diagram describing the visualization of links between artifacts in a traced project. When a user clicks the traced project, links between artifacts are recovered. A new visualization view is then created to display retrieved links in the treemap and the whole HT. When the user clicks a node in the treemap or the whole HT, a new detail view is created to display the detailed link information of the selected node in the detail HT.

3.2.3 Editing Traceability Links

We allow end users of DCTracVis to delete incorrect links and to add correct links when required, recognizing that the heuristics used to construct the links can be usefully supplemented with other information. When a node is selected in the treemap or in the whole HT, its related nodes are highlighted and a detail HT is built starting from the selected node and connecting to nodes related to it and all dependencies of these nodes. Users are then able to edit links in the treemap, the whole HT, and the detail HT views. DCTracVis provides a popup menu allowing users to delete or change existing

traceability links, add a new traceability link, and change the similarity scores ($0 \leq \text{similarity score} \leq 1$) of existing links.

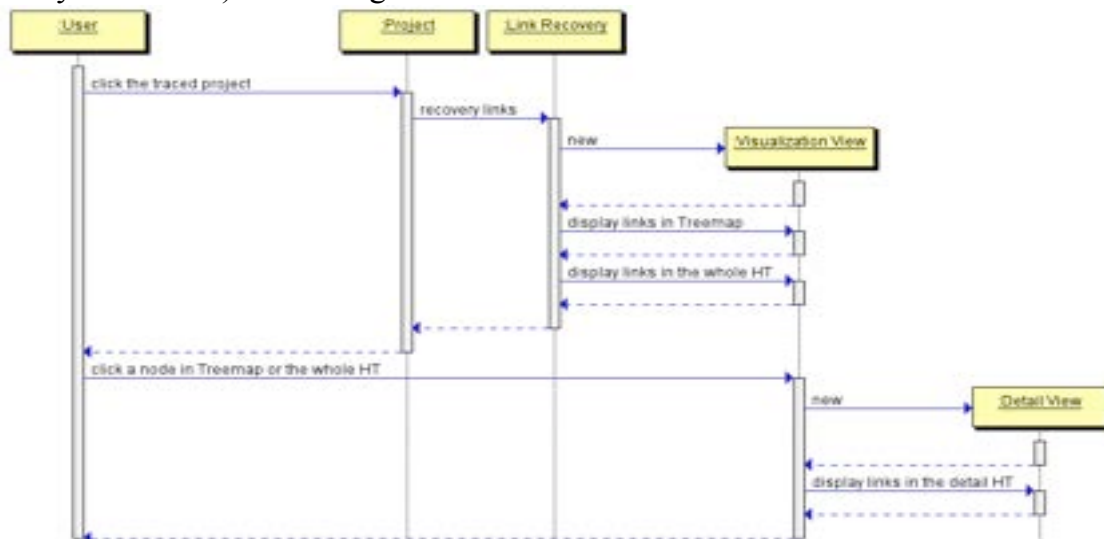


Figure 3 the Sequence Diagram for Visualizing Links in a Project

A changed link or a newly added link is assigned the highest similarity score (=1). The three views dynamically update: any change made in one view is reflected in the other two views and is saved. For instance, if an existing link is deleted in the treemap, it is deleted in the whole HT and detail HT as well, and it is not re-added if the end user runs the link extraction process again. In order to assist users in editing traceability links, we provide the full name or the similarity value when users hover the mouse over a node and the detailed content of a node when users click “Show Content” in the popup menu.

3.3 Other Functionality

Additional functionality in DCTracVis includes navigation, search, and filter support. All artifacts in the traced system are indented when listed in the navigator. This allows users to browse the list to find a specific artifact and then to locate this artifact in the treemap and the whole HT and display the detailed link information of this artifact in a detail HT. The search function enables users to use key words to find a particular node in the treemap and the whole HT. There are three methods to filter traceability links. First, users select different traceability recovery techniques to retrieve links. Second, users choose a threshold or cut point level to filter out uninteresting links. Only links that have a similarity score greater than or equal to the threshold are visualized. Third, our visualization tool allows users to filter out some uninteresting artifacts according to the number of links. Only artifacts that have more than or equal to the number of links are highlighted.

4 Usage Example

Figure 4 shows an example of the user interface of our DCTracVis prototype. Before tracing relationships between artifacts in a system and visualizing retrieved links, the DCTracVis plug-in must be installed in Eclipse, and then artifacts in the system must be

imported into Eclipse. In the “Traceability perspective”, users select the project in the “Navigation view” and click the “Start Traceability” button in the popup menu to start recovering links and then visualizing them. This screen dump (Figure 4) shows an example of retrieving and visualizing traceability links between 249 classes and 182 documentation sections in the JDK1.5 (Chen et al., 2013). Our traceability perspective includes three parts: navigation view, edit area, and traceability view. The left part is the navigation view, which displays details of a project under trace, e.g. headings inside PDF documents in the JDK1.5. The top right area is the edit area that shows java files or documents and allows users to edit them using functions provided by Eclipse IDE. The bottom right area is the traceability view that visualizes extracted links. Our visualization prototype can provide software engineers with both IDE and traceability support.

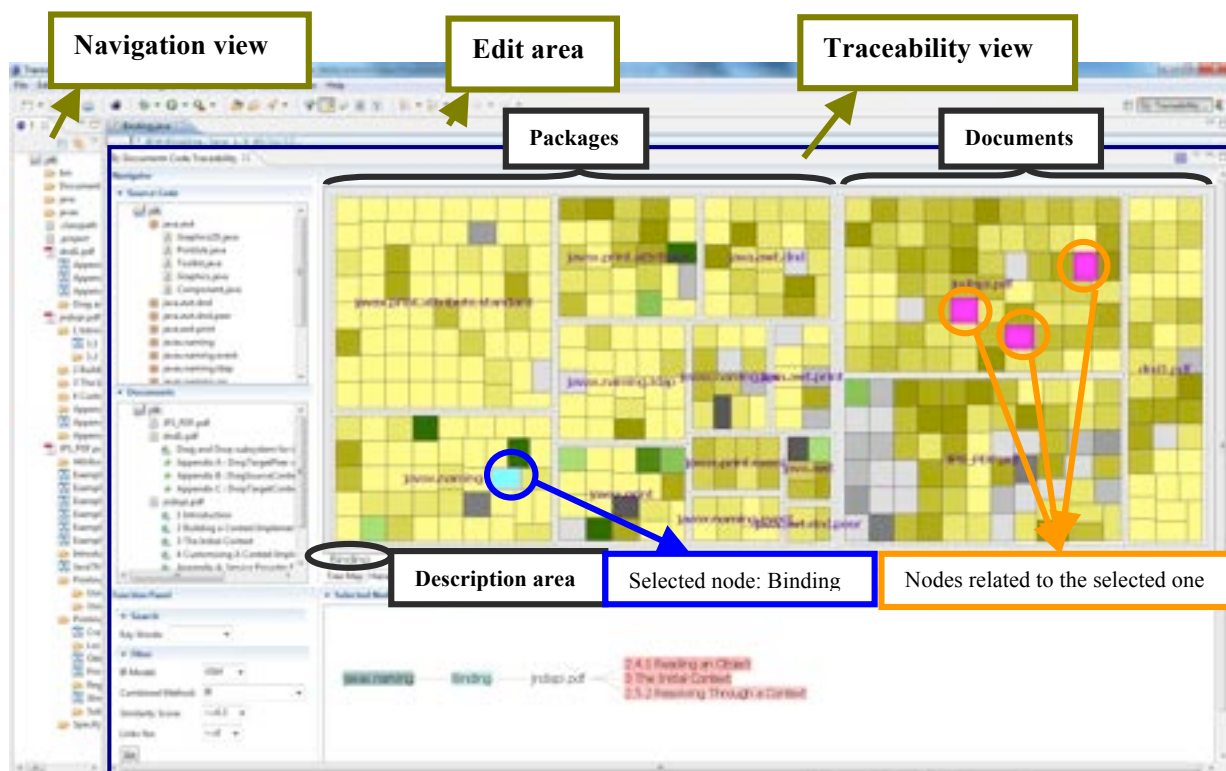


Figure 4 The User Interface of DCTracVis Using the Treemap

The traceability view includes four parts (see Figure 5). The top left area is the Navigator that lists classes and documents in the traced project in indented form to help users to find a specific item by browsing through the lists. Once an item is selected in the Navigator, its related artifacts are highlighted in the treemap and hierarchical trees. The bottom left area is the Function Panel that includes Search and Filter functions. The search function allows users to search for a specific item by key words. In Filter, users can decide whether to use the IR model alone or to combine other techniques (e.g. RE, KP and Clustering) with the IR model to capture links. Users can also select a similarity score level and a number of links level to filter out unwanted links. The top right area is the treemap view (see Figure 4) or the whole HT view (see Figure 5). The bottom right area is the detail HT view that displays the detailed information of the selected node in the treemap or the whole HT. For example, in Figure 4, a node named “Binding” with cyan colour in “javax.naming” package is selected. All related nodes are coloured

magenta in the treemap. Detailed link information is displayed in a detail HT. Simultaneously the node “Binding” in the whole HT (see Figure 5) is highlighted. Its links are shown as children of this node.

The treemap in Figure 4 is divided into two parts: one for packages and the other for documents. Classes are displayed in the packages part and sections are in the documents part. Each node is coloured using the three colour ranges (discussed in Section 3.2) according to the number of traceability links they have. When a user hovers the mouse over a node, the name of the node is described in the “Description area” at the bottom of the treemap, and all related nodes are highlighted using magenta. If the node is clicked, it is highlighted with cyan and a detail HT showing its detailed dependency information is built. For example, in Figure 4, the node “Binding” and coloured cyan in the “javax.naming” package is selected, and all related nodes are coloured magenta. Detailed link information is displayed in a detail HT (see Figure 6).

The whole Hierarchical Tree (HT) shows the whole project under trace and links in it (See Figure 5). Artifacts are displayed based on the hierarchical structure of the traced project. Classes and sections are coloured with the three colour ranges (discussed in Section 3.2) based on the number of links they have. Nodes with no links are white, to distinguish them from other nodes. The hovered-over or selected node’s name is shown in the “Description area” at the bottom of the whole HT. Traceability links of each node become their children nodes. Links are composed of two parts: the first part is names of artifacts and the second is names of items in corresponding artifacts. For example, in Figure 5, a class “Binding” in the “javax.naming” package is selected, its links are shown directly after this node and are also displayed in a detail HT (see Figure 6). Each link starts with the document’s name followed by the section’s name.

When a node is selected in the treemap or the whole HT, a detail HT is established to display detailed link information for this node (see Figure 6). The detail HT can be expanded to show link information of nodes related to the selected node (see Figure 7). Figure 7 shows that the first level is sections related to the “Binding” class, and the second level is other classes dependent on these sections. These related sections and classes are coloured to differentiate their similarity value levels. The lighter the colour the lower the similarity score a node has. When a mouse hovers over the node, its similarity score is shown. In Figure 6, the similarity value of “2.5.2 Resolving Though a Context” is 0.4. In Figure 7, the similarity score of “InitialContext” at the second level is 0.8.

Once a node is clicked in the treemap, a user can edit its links in both views. In the treemap, existing related nodes can be deleted and new nodes can be added. To prevent unwanted structural changes, names of nodes related to the selected node cannot be edited in the treemap. However, they are editable in the whole and detail hierarchical trees (see the popup menu in Figure 6); the name of an existing related node can be changed to become a new node, and their similarity scores can be changed. In all three views, we provide the contents of nodes to assist comprehension. When “Show Content” in the popup menus is selected, the file related to the node is opened in the edit area. If the node is a section, a content window is also opened to display the contents of the section. Moreover, both views are interactive; changes made in one view are reflected in the other view. For example, if an existing related node is deleted in the treemap, it is deleted in the hierarchical tree too.

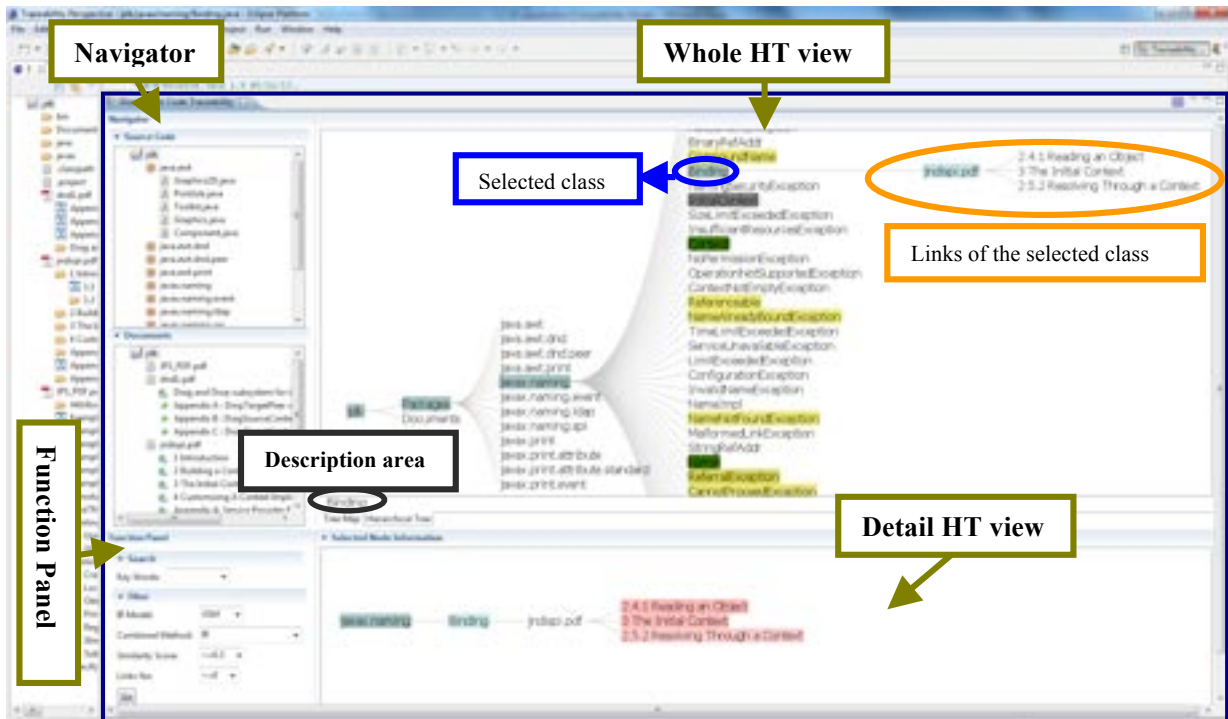


Figure 5 The User Interface of DCTracVis Using the Whole HT

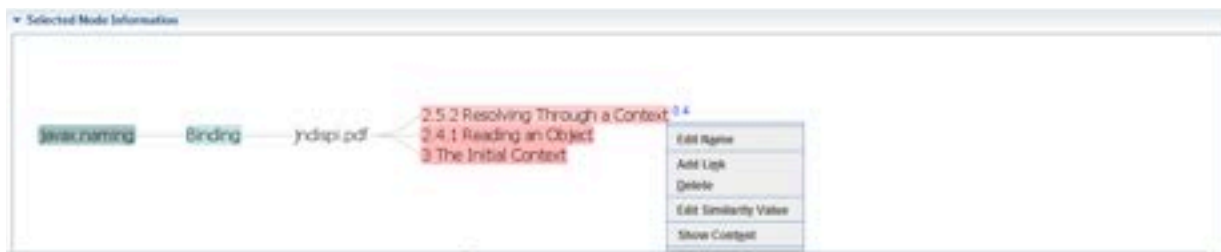


Figure 6 The Detail HT Contracted

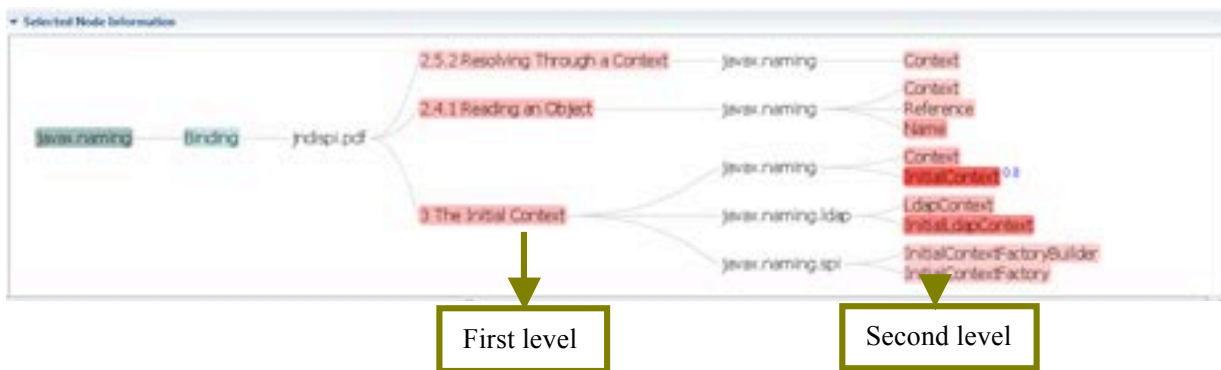


Figure 7 The Detail HT Expanded

5 Implementation

A prototype of DCTracVis has been developed. This prototype is seamlessly embedded within the Eclipse integrated development environment (IDE). It automatically extracts traceability links between sections in documents and classes in source code and visualizes these retrieved links using the treemap and the hierarchical trees.

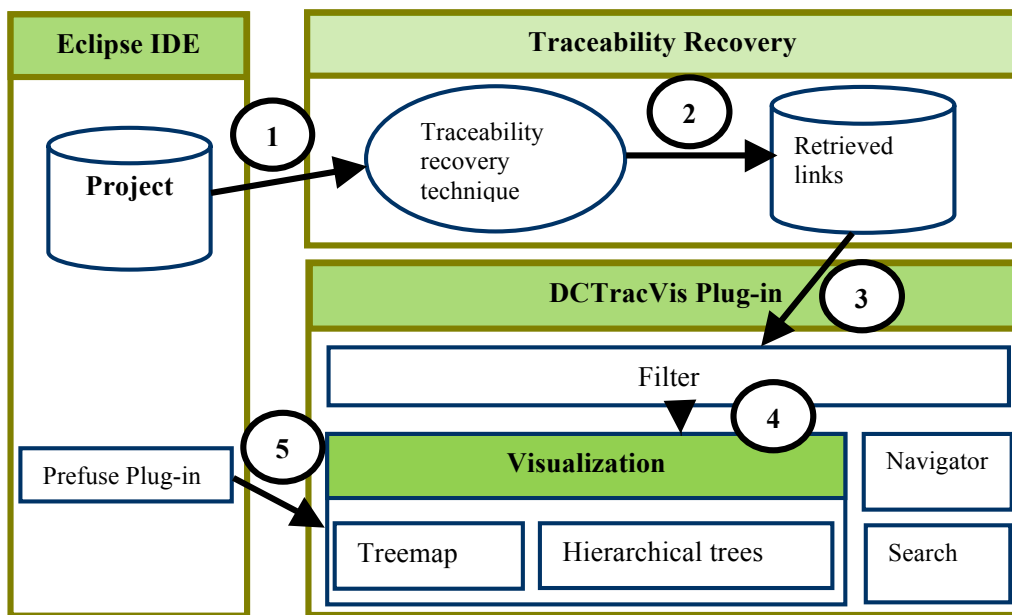


Figure 8 Architecture of DCTracVis

Figure 8 illustrates the architecture of our traceability system, DCTracVis. First, a project under trace needs to be imported into Eclipse. If documents in the project under trace contain sections, they need to be divided into smaller documents based on headings or sections. For example, if a PDF document contains 10 headings, it is split into 10 sub-documents; the contents of each are the text between its heading and the following one. Next, source code and these smaller documents are passed to our automated traceability recovery engine (1). This engine retrieves traceability links between classes and sections using a composite set of traceability recovery techniques (2).

These retrieved traceability links are then input into our traceability visualization system. They are filtered based on: (a) a threshold level, where only links with a similarity score larger than the threshold are shown to users, and (b) the number of links level, where only nodes having greater than or equal to the specified number of links are shown to users (3). After filtering, the candidate traceability links and the structure information of the project are visualized using the treemap and hierarchical tree techniques (4). Our visualization is implemented using the Prefuse Information Visualization Toolkit (Prefuse, 2011). Prefuse is an open source toolkit written in Java and supports a rich set of features for data modeling, visualization, and interaction (Prefuse, 2011). We employ Prefuse to display artifacts and links in the treemap and the hierarchical tree (5). Navigator and search functions are provided to assist users in finding a specific node.

6 Evaluation

We have conducted two evaluations to assess DCTracVis’s usability and effectiveness in comprehending, recovering, browsing, and maintaining traceability links in a traced system. Section 6.1 describes the first evaluation using four case studies and six different IR models to validate the effectiveness of our traceability recovery technique. Section 6.2

discusses the second evaluation, usability evaluation, to learn whether our traceability system supports the comprehension, browsing, and maintenance of traceability links in a system.

6.1 Evaluation for our recovery approach

6.1.1 Test cases

To validate the effectiveness of the three enhancement techniques we added to the IR models, namely Regular Expression (RE), Key Phrases (KP), and Clustering, we have set up four case studies based on four unrelated software systems: JDK1.5, ArgoUML, Freenet, and JMeter. We adopted rigorous manual identification and verification strategies (Chen and Grundy, 2011; Chen et al., 2013) to build a JDK1.5 oracle traceability link set that contains 760 correct links between classes and sections in documents. Table 3 describes the packages in JDK 1.5 and their corresponding PDF documents used in this study, as well as the number of Java classes and the number of sections in them. We divided these PDF files into sections based on their headings. The traceability benchmarks of ArgoUML, Freenet, and JMeter were kindly provided by Alberto Bacchelli (2010) comprising the three systems, their email archives, and their oracle traceability link sets that include correct links between classes and emails (Chen and Grundy, 2011). These emails were extracted from active development mailing lists of each project. Table 4 provides details of the four case studies. Here, “sections” is used for JDK1.5 and “emails” for other three cases.

Table 3 JDK1.5 Packages and Documents

JDK 1.5		#Classes / Sections
Java packages	java.awt, javax.naming, and javax.print packages	249
PDF files	JPS_PDF.pdf: Java™ Print Service API User Guide	68
	dnd1.pdf: Drag and Drop subsystem for the Java Foundation Classes	41
	jndispi.pdf: Java Naming and Directory Interface™ Service Provider Interface (JNDI SPI)	73
	Total sections:	182

Table 4 Classes, Lines of Code, Sections/Emails, Size for Documents and Total True Links Per System

System	Classes	Lines of Code	Sections/Emails	Size for Sections/Emails (MB)	Total true/correct links
JDK1.5	249	59,219	182	0.97	760
ArgoUML	423	417,811	378	1.8	308
Freenet	517	177,742	372	1.9	516
JMeter	372	172,131	348	1.7	563



Figure 9 Comparison Results between IR Only and IR+RE+KP+Clustering

6.1.2 Evaluation results

We evaluated the performance of our traceability recovery technique employing two IR engines, Apache Lucene and the Terrier IR platform. Apache Lucene uses the VSM to extract traceability links between classes and sections/emails. We also recovered links using five additional IR models, TF-IDF, BM25, DLH, PL2, and IFB2, supported by the Terrier IR platform. We applied two standard metrics – precision and recall - to measure the quality of IR models. Precision measures number of correct links over total links retrieved. Recall measures number of correct link received over total correct links.

Figure 9 illustrates the comparative results among IR models and among the combination approach (IR+RE+KP+Clustering) using different IR models in each case. We employ the same threshold scale in Chen and Grundy (2011) to illustrate precision and recall results. It contains 10 thresholds: 0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.3, 0.5, 0.7, and 0.9. A threshold < 0.3 is defined to be a low threshold in the following discussion; otherwise, it is a high threshold. The factor of using this threshold scale is to illustrate changes of precision and recall results in detail between 0 and 0.1 cut points.

Figure 9a shows that when using IR only to extract links in JDK1.5, VSM has much better performance than the other five IR models except that BM25 achieves higher precision than VSM at the 0.7 and 0.9 thresholds. The other five IR models have very similar results. However, after incorporating the three supporting techniques, RE, KP, and Clustering, with the six IR models, they produce very similar results. Precision is between 28% and 94% and recall is between 82% and 90% at all thresholds. Our combination approach using the six different IR models can achieve a very high recall (>82%) at all thresholds.

In Figure 9b, all six IR models have similar results when applying IR only to capture links in ArgoUML; low precision at low thresholds and low recall at high thresholds. However, VSM has much lower recall than the other five IR models at high thresholds. Precision and recall are changed dramatically after adding RE, KP, and Clustering to the six IR models. The precision values at all thresholds are between 19% and 59%, and recall is between 62% and 74%. Although VSM+RE+KP+Clustering achieves much better precision results (53%-58% at all thresholds), it has lower recall (around 62% at all thresholds).

For Freenet, the six IR models have similar results when retrieving links (see Figure 9c). However, VSM has lower recall than the other five IR models at high thresholds. Our combination approaches (IR+RE+KP+Clustering) using different basic retrieval IR models produce very close results. They improve precision and recall but especially recall; the recall values are between 68% and 81% at all thresholds.

When using the six IR models to extract links in JMeter, their performances are very similar; low precision at low thresholds and low recall at high thresholds (see Figure 9d). After combining RE, KP, and Clustering, their performances are very close; precision is between 10% and 54% and recall is between 72% and 80% at all thresholds.

Overall, our combination approach (IR+RE+KP+Clustering) can improve the performance of the six IR models; precision is increased at low thresholds and recall is significantly increased at high thresholds. Moreover, our approach can narrow the gap between the results of applying different IR models.

Table 5 reports the differences between the precision and recall values and the differences of the number of incorrect links (false positives) achieved with using VSM

alone versus using our approach (VSM+RE+KP+Clustering) at different levels of threshold. The results showed in Table 5 highlight that precision and recall percentages are improved at positive thresholds while recall has a slight decrease at very low thresholds. Our approach removes around 92-99% of false positives at the threshold of 0, e.g. 34604 (99%) incorrect links are reduced in ArgoUML.

Table 5 Improvement of precision, recall and reduction of number of false positives (FP) at different levels of threshold when using VSM alone versus using VSM+RE+KP+Clustering.

Improvement	Threshold 0			Threshold 0.5			Threshold 0.9		
	Precision %	Recall %	FP	Precision %	Recall %	FP	Precision %	Recall %	FP
JDK1.5	+37.79	-8.29	-9960	+8.55	+57.5	+11	+5.95	+78.29	+37
ArgoUML	+52.49	-32.79	-34604	+26.78	+52.59	+78	+10.26	+58.76	+128
Freenet	+5.24	-6.98	-22385	+42.12	+61.43	+72	+38.41	+67.24	+163
JMeter	+8.06	-8.52	-20111	+0.99	+61.1	+302	+13.14	+71.4	+352

We measure average precision to see whether our recovery approach has better performance than LSI. The results in Table 6 demonstrate that the precision value improves on average when compared to LSI results. It is noticeable that VSM outperforms LSI. These results are consistent with earlier findings (Abadi et al., 2008; Aswani and Srinivas, 2009). One possible reason for this is that there are high degrees of term overlap between queries formed by class names and relevant documents in the four cases, JDK1.5, ArgoUML, Freenet and JMeter. VSM handles well the relevant documents with high number of terms matching query terms, leaving little room for LSI to improve. However, LSI can improve relevant documents for a query with little or no term overlap. (Aswani and Srinivas, 2009)

Table 6 Average precision results for IR-alone and VSM+RE+KP+Clustering.

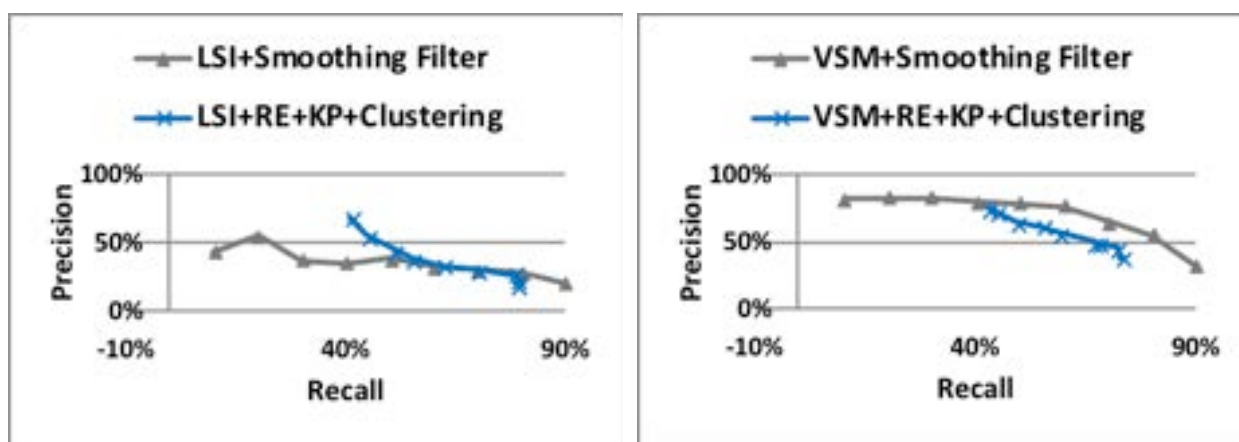
Average Precision (%)	JDK1.5	ArgoUML	Freenet	JMeter
LSI	14.87	0.31	0.25	4.93
VSM	69.21	20.22	16.59	30.21
VSM+RE+KP+Clustering	91.07	58.15	60.18	50.80

6.1.3 Comparison results

To find out whether our combination approach outperforms other approaches proposed in the literature, we compared our results to those produced in Bacchelli et al. (2010), Lucia et al. (2013), and Nishikawa et al. (2015). As ArgoUML, Freenet and JMeter have been used by Bacchelli et al. (2010), we compare our recovery approach with their light-weight methods, which are based on regular expressions. Table 7 shows that our approach provides better results than light-weight methods overall. Light-weight with an entity name that matches class entity names in emails tends to provide higher recall, light-weight with mixed approach that is matching the class file and package names in emails has higher precision, whereas our approach tends to find a better balance between precision and recall in all cases.

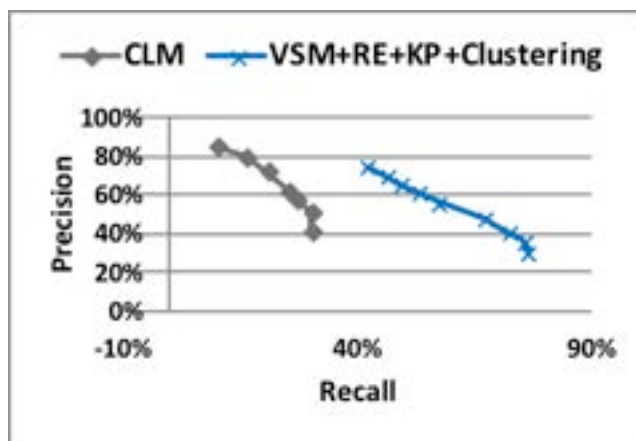
Table 7 Comparison results between light-weight methods in Bacchelli et al. (2010) and our combination approach at threshold 0.9.

Case	Recovery Approach	Precision (%)	Recall (%)	F-measure (%)
ArgoUML	Light-weight – Entity name	27	68	38
	Light-weight – Mixed approach	64	61	63
	VSM+RE+KP+Clustering	58	62	60
Freenet	Light-weight – Entity name	17	70	27
	Light-weight – Mixed approach	59	59	59
	VSM+RE+KP+Clustering	67	68	68
JMeter	Light-weight – Entity name	15	73	25
	Light-weight – Mixed approach	59	65	62
	VSM+RE+KP+Clustering	53	72	61



(a) EasyClinic: CC-TC (Smoothing Filter vs Our Approach)

(b) EasyClinic: CC-UC (Smoothing Filter vs Our Approach)



(c) EasyClinic: CLM (CC-TC(ID)) vs Our Approach (CC-TC)

Figure 10 Comparison between our Approach, the Smoothing Filter in Lucia et al. (2013), and CLM in Nishikawa et al. (2015).

To compare between our approach, the smoothing filter used in Lucia et al. (2013), and CLM used in Nishikawa et al. (2015), we use EasyClinic as the test case. EasyClinic is a software system to manage a medical doctor’s office, containing four artifacts: use cases (UC), test cases (TC), code classes (CC), and UML interaction diagrams (ID). Figure 10a compares precision/recall results between our approach and a smoothing filter

by using LSI to retrieve links between CC and TC in EasyClinic. This shows that our approach provides better precision and recall than the smoothing filter. Figure 10b is for using VSM to recover links between CC and UC; our approach produces better recall though the smoothing filter has higher precision. Figure 10c is the comparison between our approach and CLM. The result indicates that our approach outperforms CLM, producing higher recall. In summary, our approach can provide better recall than the smoothing filter and CLM.

6.2 Usability evaluation

We undertook a usability study of DCTracVis to answer the question: whether our approach of combining treemap and hierarchical tree views help to support the comprehension, browsing, and maintenance of traceability links in a system. The case used in this study is the JDK1.5 (see Tables 3 and 4), which contains 760 true links between 249 classes and 182 sections/emails.

6.2.1 Study design

To answer the question, this study contained two parts. Part 1 was using Eclipse and the PDF Reader tool to complete three tasks. Part 2 was to employ our tool (DCTracVis) to complete the same three tasks. The three tasks were: 1) to understand the JDK1.5 system; the structure of the system and the overview of links between artifacts in the system. 2) to understand how an artifact works; how a class works in order to fix a bug related to it, where the documentation of this class can be found, and what other classes are related to this class. 3) to modify traceability links of an artifact; links of a class retrieved by IR recovery techniques may contain incorrect links or may miss correct links or may have low similarity scores for correct links; these retrieved traceability links of the class need to be edited to contain only correct links by deleting incorrect links or adding missing links. During each part, a brief introduction and a demonstration were provided to help participants to gain familiarity with Eclipse, the PDF reader and our tool at the beginning. The participants then performed the three tasks. In Part 2, after completion of the study tasks, our participants also needed to answer questions that were designed to reflect user perceptions of our tool.

6.2.2 Study participants

We recruited 40 participants for the evaluation of DCTracVis. We randomly divided them into two groups: a control group for Part 1 and an experimental group for Part 2. In the experimental group, the participants were 13 university students and 2 academics and 5 from industry (see Figure 11a). The Control group has more academics (5) and industry (7) but less students (8). Figure 11b shows the software development experience each participant had. Among 20 participants in the experimental group, 3 had more than 10 years of development experience, 7 had fewer than 10 years but more than 5 years, 8 had fewer than 5 years but more than 1 year, and 2 had less than 1 year. But the majority of participants in the control group have more than 5 years of development experience. All participants in the experimental group had at least some experience with the use of Eclipse for programming Java systems (see Figure 11c). Among them, 3 always used

Eclipse for software development, 5 usually used Eclipse, 8 sometimes used it, and 4 rarely used it. For the control group, most of them were familiar with Eclipse. In the experimental group, only one participant usually applied traceability tools to assist in comprehending or maintaining or programming software systems (See Figure 11d). This was also the case in the control group.

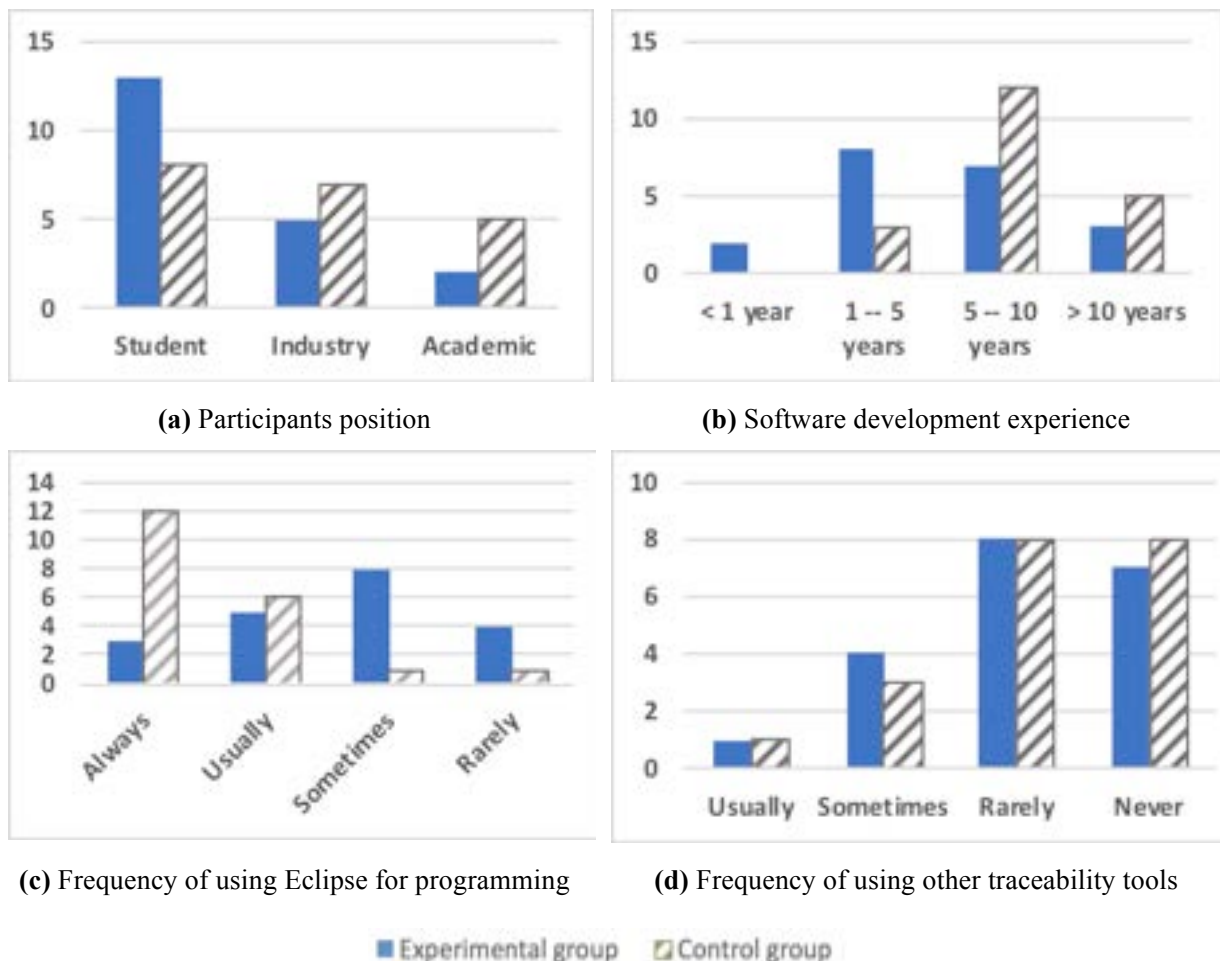


Figure 11 The Background Information of Participants

6.2.3 Study results

Table 8 summarizes what features were used by participants to complete each task. The first interesting result was revealed when performing the first task (understanding the JDK1.5 system). In the experimental group, 13 of the 20 participants completed this in less than 1 minute, 4 spent 2 minutes, 1 spent 3 minutes, 1 spent 4 minutes, and the longest took 9 minutes to complete this task. Based on our observations, the participant who took 9 minutes spent most of their time gaining familiar with our tool as they had done little practice before performing the three tasks. However, they strongly agreed that DCTracVis clearly illustrated the hierarchical structure of the system and provided a good overview of links in the system. They accomplished this task by using the navigator, filter, treemap, and/or whole hierarchical tree functions (See Table 8). In the control group, the participants spent around 7 minutes on average to complete the first task. Each

participant used navigator, search and code reading and documents to understand the JDK1.5 system.

Table 8 Features usage statistics in experimental and control groups

Function	Group	Task1	Task2	Task3
Navigator	Experimental	9	12	12
	Control	20	20	20
Search	Experimental	0	6	8
	Control	20	20	20
Filter	Experimental	20	0	0
	Control	/	/	/
Class contents	Experimental	0	20	10
	Control	20	20	20
PDF contents	Experimental	0	20	12
	Control	20	20	20
Treemap	Experimental	6	1	4
	Control	/	/	/
Whole hierarchical tree	Experimental	10	1	3
	Control	/	/	/
Detail hierarchical tree	Experimental	0	20	18
	Control	/	/	/

All participants of the experimental group completed the second task (bug fixing) with times varying from 1 minute to 5 minutes, whereas, in the control group, participants took 4 to 13 minutes to complete. We noticed that participants in the control group took great effort in detecting links by taking notes and frequently switching between Eclipse and PDF reader to understand code and PDF contents. On the contrary, all participants in the experimental group completed this task quickly and easily. 12 participants used the navigator function to find a specific node, 6 used the search function, 1 directly located the node in the treemap, and 1 used the whole hierarchical tree (see Table 8). All participants used the show class and PDF content function to help them understand links and utilized the detail hierarchical tree to accomplish this task. They agreed that the detailed dependency information provided in the detail hierarchical tree view was a good supplement to both the treemap view and the whole hierarchical tree view while performing the second task.

The third task (link modification) was completed within 2 and 6 minutes for the experimental group, while the control group took 5 to 15 minutes. In the experimental group, 12 participants located a node using the navigator function, and 8 used the search function (see Table 8). The majority of participants (18 of 20) undertook the modification of traceability links of a node using the detail hierarchical tree view. Because they thought this view was more intuitive and straight-forward for this task. 2 used either the treemap or the whole hierarchical tree to modify links of a node. 3 participants edited links of a node in the treemap and the detail hierarchical tree, and 2 completed the link modification in the whole hierarchical tree and the detail hierarchical tree. 12 participants used the show content function to support them in comprehending the modified links. 19 participants could quickly modify links. However, one participant looked frustrated and stressed while doing the link modification. They complained that the edit menu was not easy to use. This tough situation happened to all participants in the control group because they had to go through all PDF files to locate the correct links.

In the experimental group, most participants used the navigator function to seek a specific node in the second and third tasks as they thought it was easier and more natural to browse artifacts in the navigator to find the node they were interested in. We noticed that the participants encountered difficulties in directly finding a specific node in the treemap. However, with the support of the navigator and search functions, they easily and quickly found an item they were interested in. Our observation also indicated that three participants felt confused after applying the filter. They suggested it would be better to provide a summary of how many links were recovered and how many links filtered out after using the filter. In the control group, participants had to put in more effort to accomplish the three tasks. They used navigator in Eclipse to find classes, used the search function to find links in PDF documents, and had to read code and documents to edit links, which is a time-consuming and inefficient process. They commented that for “tedious tasks, tool support would be so so good.”

6.2.4 Information load results

Information load refers to the amount of information available for judgement; participants can view the information in one interface or in more than one interface (Kardes et al., 2004). We compared the information load of the two groups from two factors: information density and context switching.

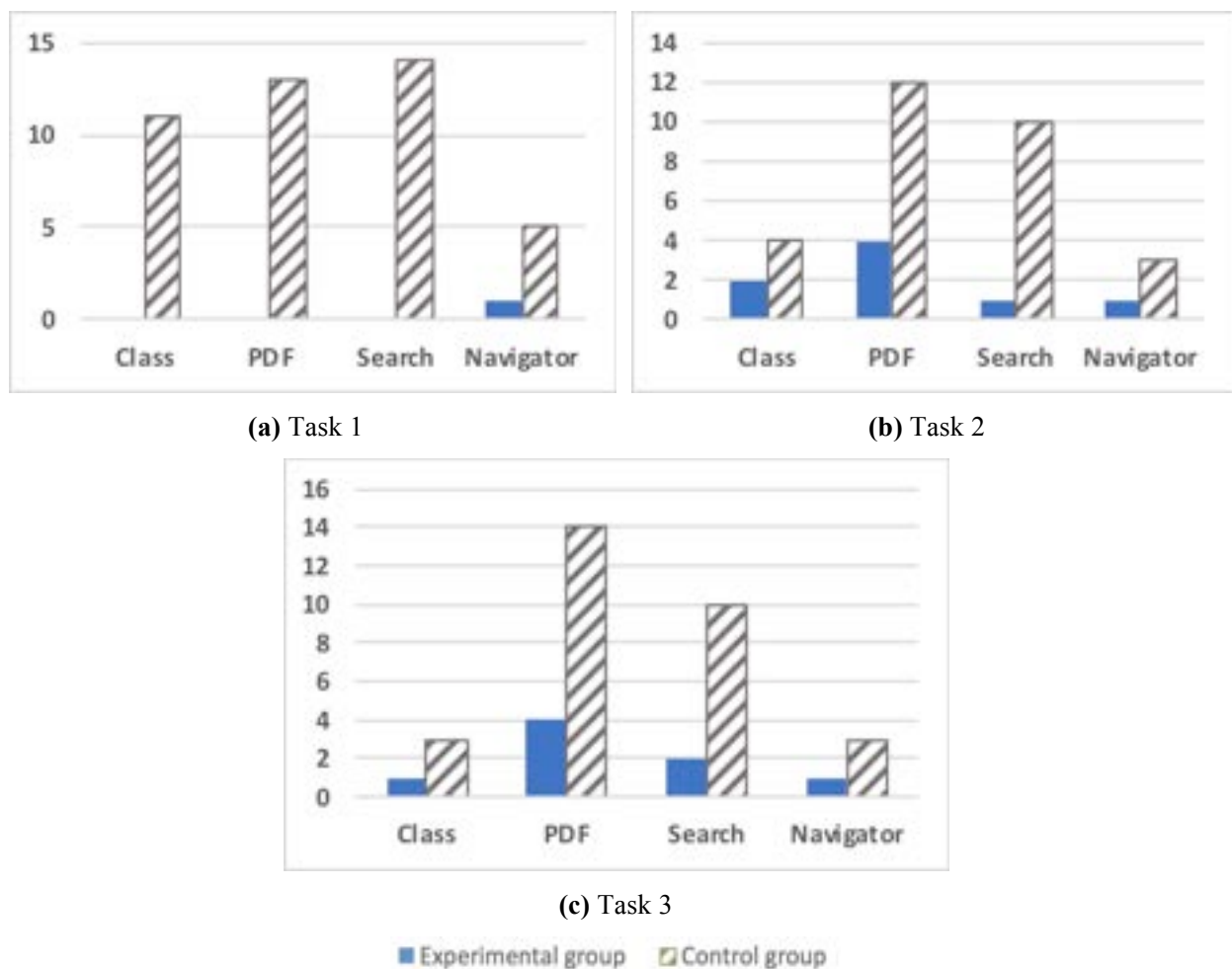


Figure 12 Average Participant Actions for Various Features in Experimental and Control Group

Information density examines the compactness of an interface in terms of the amount of information conveyed (Woodruff and Stonebraker, 1997). We observed that participants using our tool took less actions in performing the three tasks compared to participants using Eclipse and PDF reader. Figure 12 shows the average number of actions participants took during the completion of the three tasks. The results indicate that participants took less actions hence less effort with our tool. Our tool can provide higher volumes of information effectively in a single view.

Context switching involves storing the old state and retrieving the new state, switching from one interface to another (Li et al., 2007). To complete the three tasks, all participants in the control group frequently switched between Eclipse and PDF reader to read code and documents. Participants in the experimental group just stayed in our tool to view links and the content of code and documents. We also noticed that all participants in the control group had to take notes to keep a mental track of all links they found. By comparison, participants in the experimental group could browse links in the treemap or the hierarchical tree easily. Moreover, participants in the control group spent more time and cognitive efforts in finding correct links. However participants in the experimental group could find and edit links quickly and easily. Most participants in the control group commented that it would be good to have an automatic traceability tool to help in easy link maintenance.

6.2.5 Questions results

The main results analysis performed after this evaluation was on the set of questions answered by participants based on their experiences of using DCTracVis in comparison to other software tools that they have used. We start with the analysis of the evaluation in the functionality of DCTracVis. The results can be seen in Figure 13. The diagram shows the seven questions (effective link recovery, easy to extract links, easy to maintain (add, delete, edit) links, easy to browse links, easy to find links, support comprehension, and support maintenance/development) on the x-axis. The y-axis shows the number of participants; how much they agreed (strongly agree, agree, or neutral, disagree, or strongly disagree) that our tool helped recover, browse, and maintain traceability links in a system and supported users in comprehending, maintaining, or developing the system.

No participant gave a negative response to any of the seven questions. All participants strongly agreed or agreed that our tool was easy to use to extract traceability links, easy to browse links, and easy to find links. 19 participants (strongly) agreed that it helped them more effective in extracting links between artifacts within systems. Results in Chen et al. (2013) concerning manually building the oracle link set further reinforce the claim that our tool can effectively and easily extract links. Our tool only takes up to 3.5 minutes to recover links between 249 classes and 182 sections in JDK1.5 when running on an iMac with a 2.4GHz Intel Core Duo processor and 3GB of RAM, and combining different IR models. Our tool achieves high recall, between 82% and 90%, at all thresholds (See Figure 9). However, when we manually built the JDK1.5 benchmark, every participant spent one hour to identify the related sections for 50 classes on average during the link recovery. Furthermore, it took a longer time to do the link verification than the link recovery on average. (Chen et al. 2013)

For link maintenance, 18 (strongly) agreed that our tool was easy to maintain links. 18 participants (strongly) agreed that it was useful to support them in the comprehension and

maintenance/development of a system. Several participants gave a neutral answer to questions for effective link recovery, link maintenance, comprehension, and maintenance and development. They responded this way because they could not undertake the comparison as they had never used other traceability tools.

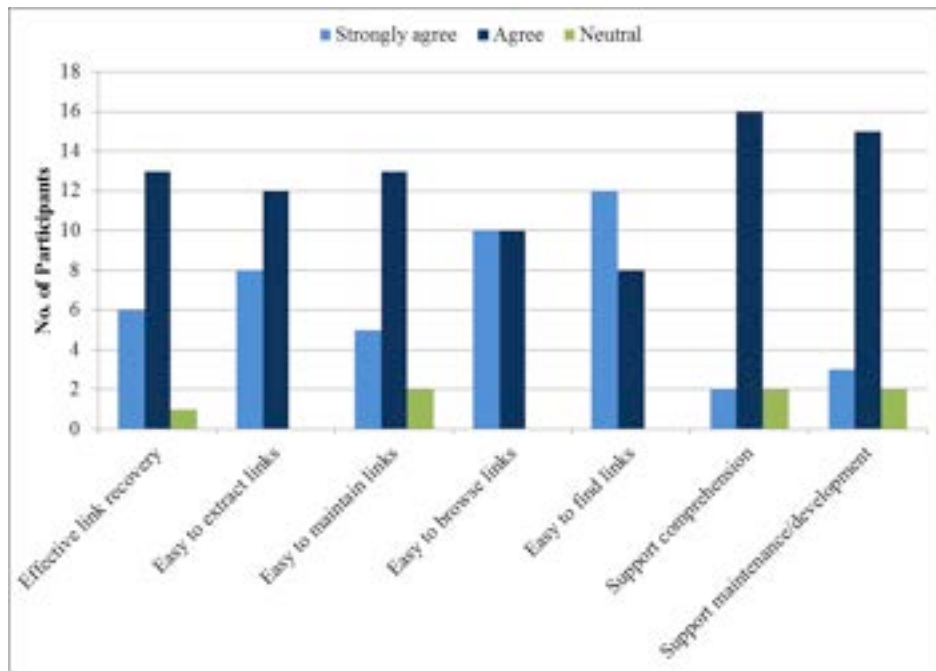


Figure 13 Results of the Evaluation in the Functionality of DCTracVis

Next, we analyzed the evaluation of the overall performance of DCTracVis. Figure 14 shows the evaluation results. The x-axis displays the eight questions (functions well integrated, functions all present, user friendly, easy to use, learn quickly, easy to learn, like to use it in the future, and recommend to friends) concerning our tool’s overall performance. The y-axis, as previously, shows the number of participants and their agreement level.

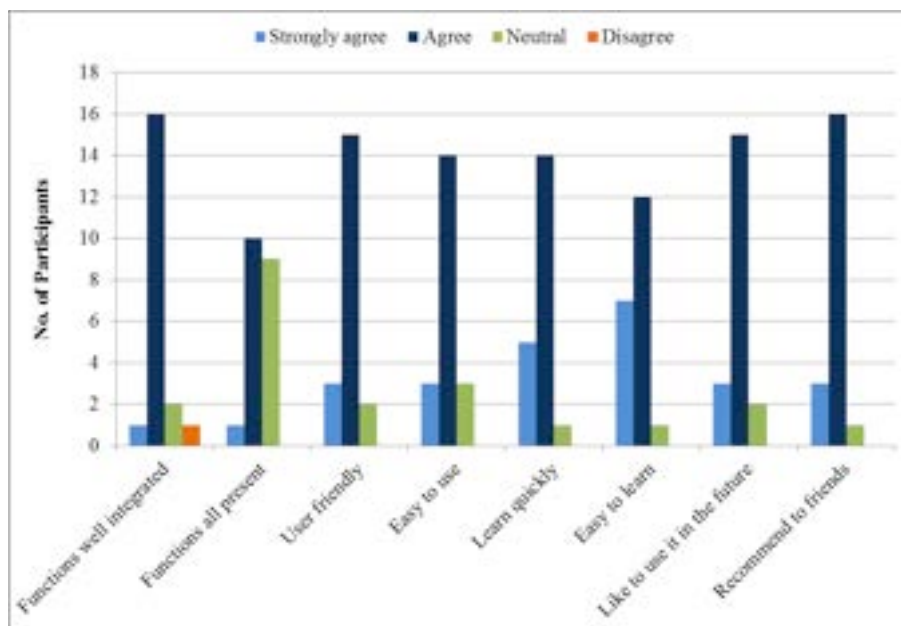


Figure 14 Results of the Evaluation in the Overall Performance of DCTracVis

One participant disagreed that the various functions in our tool were well integrated and easy to find. They found it was a little difficult to modify links of a node using the edit menu. 19 participants (strongly) agreed that they learned to use our tool quickly, it was easy to learn how to use it, and they would recommend it to friends. 18 of them (strongly) agreed that our tool was user friendly, and they would like to use it in the future. 17 participants (strongly) agreed that it was easy to use, and functions in it were well integrated and easy to find. 11 agreed that all the functions they expected were all present, but 9 participants gave a neutral answer to this question. 5 participants thought that there was room to improve although expected functions were all present. 4 participants responded this way because they could not undertake the comparison as they had never used other traceability tools. Several participants provided a neutral feedback to the other seven questions. They commented that they were unable to conduct the comparison because they had no experience of using other traceability tools.

The participants also reported many valuable comments about our tool via the questionnaire's open answer questions. These comments include the following:

- **Browsing and Maintenance:** In terms of colors, one participant pointed out that it was not feasible for colour-blind users to discern nodes if we adopted inappropriate colours to represent the number of links that each node has in the treemap or the whole hierarchical tree and to differentiate the similarity value level of each link in the detail hierarchical tree. Five participants also commented that colours used to identify the number of links in the treemap might confuse users, and it was hard to remember them. One participant suggested that the treemap should be divided into two sections using different colours. Moreover, two participants commented that it was not easy to quickly notice the selected node and its related nodes. In terms of other representations, five participants commented that it would have a better visual effect if the zooming was improved, and the tool made use of a large screen or large view windows, or allowed view windows to be editable. Five participants suggested that it would be helpful to use different sizes of nodes to represent the number of links that each node has, or to reflect the sizes of classes and sections in the system, or to differentiate the similarity score levels. Two participants commented that it needed to be more user friendly or fancier.
- **Comprehension:** Two participants suggested that words related to a selected node should be highlighted when showing the contents of the related node. Two participants thought that it would be helpful to provide summary documents about the meta-data of the traced system, such as how many packages, documents, classes, retrieved links, related links of each node, and so on.

Finally, we wanted to learn which visualization view the participants preferred with regard to browsing links, maintaining links, and supporting them in the comprehension and maintenance and development of a system. Figure 15 shows the evaluation results of a comparison made between the treemap view and the whole hierarchical tree view. More participants preferred the whole hierarchical tree to the treemap in browsing links (12 vs. 8). 11 participants preferred the treemap to maintain links and to support them in understanding the traced system, but 9 participants preferred the whole hierarchical tree. For the support of maintenance and development of a system, 10 preferred the treemap

and the others preferred the hierarchical tree. Overall, 12 participants selected treemap as the best visualization, while the others (8) preferred the hierarchical tree.

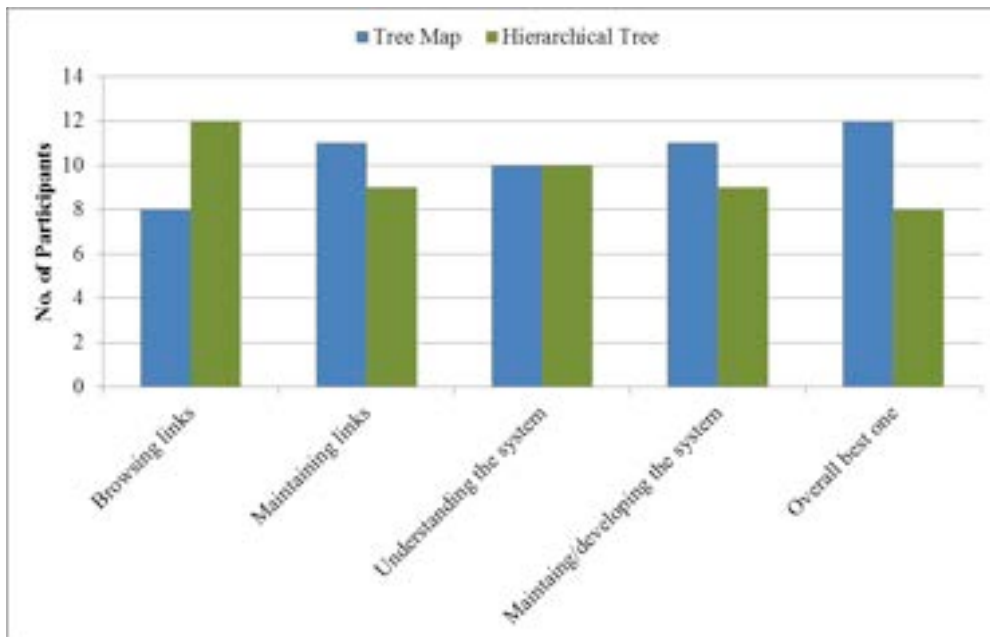


Figure 15 Results of Comparison among the Two Visualization Views

The main reasons that the participants thought the treemap was the best visualization view included the following.

- **Browsing and Maintenance:** 4 participants pointed out that the detail hierarchical tree was a good supplement to the treemap, and their combination was easy to use. 2 participants also reasoned that the treemap was easy to move around and easy to map the relationships between documents and classes. 2 participants ranked the treemap as the best based on colour and overall view, and it was more useful for maintaining as well as browsing than the whole hierarchical tree.
- **Comprehension:** 2 participants commented that the treemap presented more information than the whole hierarchical tree without being as cluttered as the tree view, and that it showed the complete system and allowed them to look into details.

The main reasons that the participants disliked the treemap included the following:

- **Browsing and Maintenance:** 1 participant mentioned that the treemap was a quite good-looking, rich-user-experience but it was not easy to perform tasks such as navigating to a class, or editing or deleting a link.
- **Comprehension:** 1 participant commented that the treemap was compact but class names were not shown inside the map, and also the maps for the packages and document were identical, making it difficult to remember which map was for packages or documents.

The reasons that participants preferred the whole hierarchical tree included:

- **Browsing and Maintenance:** 4 participants commented that the whole hierarchical tree was easier to use and easy to browse links.

- **Comprehension:** 2 participants thought that the whole hierarchical tree was more intuitive and comprehensive than others.

One participant commented that a disadvantage of the whole hierarchical tree was that users had to browse through the hierarchical tree step-by-step to find a node. Overall, both the treemap and the whole hierarchical tree have their strengths and weaknesses. 6 participants suggested that there was room to improve the visualization view to be fancier and to provide more animations. One participant also commented that software developers preferred to use a class diagram as it was the traditional way for developers to understand the system, and it was easy to make sense of relationships between classes.

7 Discussion

According to the evaluation results, the three enhancement techniques demonstrate the capability of improving the performance of IR models. The most obvious observations from the all of the case studies are: precision at low thresholds is increased, recall is significantly improved at high thresholds, and the results of applying different IR models become less differentiated. Our approach produces a much better result than an IR model alone but it takes a longer time to process. We ran our approach on an iMac with a 2.4 GHz Intel Core Duo processor and 3GB of RAM. Our approach took up to 5 minutes to execute on each case with different IR models. Around 80% of the time is spent on Key Phrases extraction. This is because KEA, the Key Phrases processor in our approach, uses an expensive machine learning algorithm for training and key phrase extraction (Witten et al., 1999). Our approach is much faster than manually capturing links: when building the oracle link set for JDK1.5, participants spent one hour to identify related sections of 50 classes on average (Chen and Grundy, 2011; Chen et al., 2013).

Our usability evaluation of DCTracVis obtained positive results. The participants could recover traceability links in a traced system effectively and efficiently. The participants also could easily browse links and find a specific node. Furthermore, the participants could easily and conveniently modify links of a node. In addition, our tool supported the comprehension of links.

The usability evaluation showed that each of the treemap and the hierarchical tree have their advantages and disadvantages. 12 participants were satisfied with the performance of the treemap. 8 thought that the hierarchical tree outperformed the treemap. We combined the treemap and the hierarchical tree to visualize links in the system and adopted another hierarchical tree to display the detailed link information of a node. Our visualization approach took advantage of the strengths of each of them to ameliorate limitations of each of them. Our tool provided multiple approaches to visualize links to meet different participants' needs. Our tool allowed the participants to easily gain the structure of the traced system and the overall overview of links in it.

However, the usability evaluation exposed some weaknesses with our tool. We propose some possible solutions and improvements for these weaknesses.

- **Browsing and Maintenance:** 4 participants had trouble in recognizing/remembering colors used in the detail hierarchical tree. It would be more intuitive to employ different font sizes and/or colors of nodes in the detail hierarchical tree to display their similarity value levels and to make the important

links more visible. 2 participants felt that the selected node and its related links were not noticeable. This could be addressed by enlarging selected node and their related nodes to make them stand out from other nodes in the treemap. 2 participants also felt that contents in the content window were hard to read. Highlighting words that are related to the selected node could address this. Moreover, 6 participants encountered difficulty remembering colors that differentiated the number of links level. To apply or combine other methods to represent the relationship status of each node in the treemap and the whole hierarchical tree would make the view more intuitive. In addition, 11 participants appeared to have high expectations regarding the ability to edit using the windows of the navigator, the filter, the treemap, the whole hierarchical tree, and the detail hierarchical tree, and the ability to move them around. It would be more visually effective to separate them into different independent and editable view windows.

- **Comprehension:** 2 participants appeared to be disappointed that our tool didn't provide a summary report about the traced system after adopting the filter. It would be more understandable to supply a summary report to briefly introduce the traced system whenever the filter is used.

These propositions all represent potential future work for refining our tool. Some other interesting future work includes the following: (1) In order to retrieve more correct links and fewer fault links, allow users to edit the regular expressions used to match words in documents; (2) Allow users to add new key phrases or edit/delete existing key phrases to refine extracted key phrases to recover more related links; (3) Include a history navigation, which allows users to learn the history of their movements and activities and to undo or redo previous activities. (4) Allow modifications made to traceability links to be saved. (5) Examine other key phrase extraction techniques to improve execution time. (6) Other enhancement techniques to improve precision and recall of IR models. (7) Use other IR models to evaluate and validate our recovery approach to improve the performance of IR models.

There are four main limitations of our tool. (1) The size of each node in the treemap becomes small in order to display a system with large numbers of artifacts in one screen. (2) The three color ranges used in the treemap and the whole hierarchical tree may need to be extended to clearly distinguish nodes if the range of numbers of links that nodes have becomes large. Allowing user configuration of colours and colour gradients may be helpful, especially for colour-blind users. (3) Some correct links are discarded after adding clustering because correct links that are not included in clusters are cut out (Chen and Grundy, 2011).

8 Threats to Validity

There are some threats that could affect the validity of the evaluation of our traceability link recovery approach and the usability study of our tool, DCTracVis.

For the evaluation of our link recovery technique, there are two threats. First, we relied on human judgment to build the oracle link set and thus this set might not 100% correct. To alleviate this, in the case of the JDK1.5 oracle, we applied a very rigorous manual verification strategy to analyze every true link, which were verified by at least 2 analysts

(Chen et al., 2013). The other three benchmarks, ArgoUML, Freenet and JMeter, have been used by other authors (Bacchelli et al., 2010). Second, our traceability recovery technique may show different results when applied to other software systems with other types of documents. To alleviate this, we chose 4 unrelated open-source systems. These systems vary in the size of the systems, the types of documents, the structures of documents, and the availability of comments in source code. However, we cannot confirm that our results are similar in closed-source systems.

For the usability study of DCTracVis, we used only one system, JDK1.5. Hence, we cannot claim that these results would work for all systems. To mitigate this threat, we made sure we used an open source, widely used system. The second threat is our participants were limited to a small subset of students and developers who volunteered, and therefore might be biased towards assessing DCTracVis. However, we made sure to recruit participants who had different experience and familiarity in traceability and software development. We also recruited participants from industry and academia, since practices in the system development and maintenance may vary from those used by students. The final threat to validity is caused by using participants that were unfamiliar with the JDK1.5 and the concept of traceability. However, in a real-world scenario, system maintenance is often performed by developers who are not as familiar with the system as the original developer. To resolve this, at the beginning of the study we gave a detailed description about the JDK1.5 and traceability, provided a demo and demonstrated our tool, and gave participants sufficient time to be familiar with JDK1.5, traceability and our tool.

9 Conclusions

It is a major challenge for automated IR traceability recovery techniques to extract links between artifacts of a system at high-levels of precision and recall no matter which threshold is chosen. The shortcoming of commonly used IR models is their low precision with high recall at low thresholds or high precision with low recall at high thresholds. We have developed a new recovery approach by incorporating three enhancement techniques, RE, KP, and Clustering, with IR to extract links between documents and class entities. Our approach ameliorates the limitations of IR by taking advantage of the strengths of these three enhancement techniques. Our experimental results based on four case studies and applying six different IR models provides a demonstration that the limitations of IR can be effectively mitigated by combining these three enhancement techniques. Our approach provides reasonable precision and high recall at all thresholds.

After capturing traceability links, it is a large challenge to visualize them effectively and efficiently because of scalability and visual clutter issues. We present an approach that integrates enclosure and node-link visualization representations to support the overall overview of traceability in the system and the detailed overview of each link while still being highly scalable and interactive. The treemap and hierarchical tree visualization techniques are applied to display traceability links in a system. The treemap view provides the overall structure of the system and the overall overview of traceability links. Our approach reduces visual clutter through adopting colors to represent the relationship status of each node instead of directly drawing edges between related nodes on top of the treemap. Two hierarchical trees are employed to visualize links. The whole HT view is to

represent the whole system under trace and links in it to communicate the hierarchical structure of the system. The detail HT view can be treated as a supplement to the treemap and the whole HT. When a node is selected in the treemap or the whole HT, the detail HT view displays all nodes that are related to the selected node and other dependency information of these nodes. These traceability links can be modified (add, delete, edit) and their similarity scores can also be changed. The three views are dynamically updated, changes made in one view are reflected in the other two views, and vice versa.

We adopted this composite recovery approach and composite visualization approach to build a novel link traceability system, DCTracVis. This provides support including navigation, search, and filter functions to assist users in locating a specific node or filtering out some uninteresting links. Our usability study shows that our traceability system performed well and was both helpful and useful. Our system was able to extract traceability links in a system easily and effectively. Our system also allowed users to easily browse links and to quickly locate a specific link. Moreover, it allowed users to easily and conveniently maintain links. In addition, it supported the comprehension of links and provided the hierarchical structure of the system and the overall overview of links.

Acknowledgement

The authors gratefully acknowledge Alberto Bacchelli for agreeing to share his exemplar test sets and oracles of ArgoUML, Freenet and JMeter.

References

- Abadi, A., Nisenson, M., and Simionovici, Y.: A traceability technique for specifications. 16th IEEE International Conference on Program Comprehension, pp. 103-112 (2008)
- ADAMS 2009. Overview. Data accessed: February 2009, <http://adams.dmi.unisa.it/adams-2009/Overview.html>
- Antoniol, G., Casazza, G., and Cimitile, A.: Traceability recovery by modelling programmer behavior. 7th WCRE, Queensland, Australia, Nov., pp. 240-247 (2000)
- Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D., and Merlo, E.: Recovering traceability links between code and documentations. TSE, Vol. 28, No. 10, Oct., pp. 970-983 (2002)
- Asuncion, H. U., Francois, F., and Taylor, R. N.: An end-to-end industrial software traceability tool. ESEC-FSE, Cavtat near Dubrovnik, Croatia, pp. 115-124 (2007)
- Aswani Kumar, Ch. and Srinivas, S.: On the performance of Latent Semantic Indexing-based Information Retrieval. Journal of Computing and Information Technology – CIT 17, 2009, 3, pp. 259-264 (2009)
- Bacchelli, A., Lanza, M., and Robbes, R.: Linking E-mails and source code artifacts. ICSE'10, May, pp.375-384 (2010)
- Cleland-Huang, J. and Habrat, R.: Visual support in automated tracing. REV 2007, IEEE Computer Society, pp. 4-8 (2007)
- Cleland-Huang, J., Settimi, R., Duan, C., and Zou, X.: Utilizing supporting evidence to improve dynamic requirements traceability. RE'05, Paris, pp.135-144 (2005)
- Cleland-Huang, J., Settimi, R., Romanova, E., Berenbach, B., and Clark, S.: Best practices for automated traceability. Computer, Vol. 40, Issue 6, pp. 27-35 (2007)

- Chen, X. and Grundy, J.: Improving automated documentation to code traceability by combining retrieval techniques. ASE 2011, pp. 223-232 (2011)
- Chen, X., Hosking J., and Grundy, J.: Visualizing Traceability Links between Source Code and Documentation. VL/HCC 2012, Innsbruck (2012)
- Chen, X., Hosking, J., Grundy, J., and Amor, R.: Development of robust traceability benchmarks. 22nd ASWEC, June 2013, Melbourne, Australia (2013)
- Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J.J., and van Deursen, A.: Understanding execution traces using massive sequence and circular bundle views. 15th ICPC, Alberta, BC, pp. 49-58 (2007)
- Falessi, D., Di Penta, M., Canfora, G., and Cantone, G.: Estimating the number of remaining links in traceability recovery. Empirical Software Engineering (EMSE), June 2017, Volume 22, Issue 3, pp. 996-1027 (2017)
- Gotel, O.C. and Finkelstein, A. C. W.: An analysis of the requirements traceability problem. 1st RE, pp. 94-101 (1994)
- Graham, M. and Kennedy, J.: A survey of multiple tree visualization. Information Visualization, Vol. 9(4), pp. 235-252 (2010)
- Grechanik, M., McKinley, K. S., and Perry, D. E.: Recovering and using use-case-diagram-to-source-code traceability links. ESEC/FSE'07, Croatia, pp. 95-104 (2007)
- Hayes, J. H., Dekhtyar, A., and Osborne, J.: Improving requirements tracing via information retrieval. Proc. Int'l Conf. Requirements Eng. (RE), pp. 151-161 (2003)
- Holten, D.: Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. IEEE Transactions on Visualization and Computer Graphics, Vol. 12, No. 5, pp. 741-748 (2006)
- Jackson, M. and Wilkerson, M.: MBSE-driven visualization of requirements allocation and traceability. Aerospace Conference, MT, USA, pp. 1-17 (2016)
- Kamalabalan, K., Uruththirakodeeswaran, T., and Balasubramaniam, D.: Tool support for traceability of software artefacts. Moratuwa Engineering Research Conference (MERCon), Moratuwa, Sri Lanka, pp. 318-323 (2015)
- Kardes, F. R., Cronley, M. L., Kellaris, J. J., and Posavac, S. S.: The role of selective information processing in price-quality inference. Journal of Consumer Research, 31(2), pp. 368-374 (2004)
- Kuang, H., Nie, J., Hu, H., Rempel, P., Lu, J., Egyed, A., and Mader, P.: Analyzing closeness of code dependencies for improving IR-based traceability recovery. IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER), Feb. 2017, Klagenfurt, Austria, pp. 68-78 (2017)
- LDRA, LDRA Product Brochure v7.2. 2012. From <http://www.ldra.com>
- Li, C., Ding, C., and Shen, K.: Quantifying the cost of context switch. Proceedings of the 2007 workshop on experimental computer science (ExpCS'07), San Diego, California, June (2007)
- Li, Y. and Maalej, W.: Which traceability visualization is suitable in this context? A comparative study. REFSQ 2012, LNCS 7195, pp. 194-210, (2012)
- Lucia, A. D., Fasano, F., Francese, R., and Tortora, G.: ADAMS: an artifact-based process support system. 16th Int. Conf. on Software Engineering and Knowledge Engineering, Alberta, Canada, pp. 31-36 (2004)
- Lucia, A. D., Fasano, F., Oliveto, R., and Tortora, G.: Recovering traceability links in software artifact management systems using information retrieval methods. TOSEM, Vol. 16, No. 4, Article 13 (2007)

- Lucia, A. D., Penta, M. D., Oliveto, R., Panichella, A., Panichella, S.: Applying a smoothing filter to improve IR-based traceability recovery processes: an empirical investigation. *Information and Software Technology*, 55(2013), pp. 741-754 (2013)
- MacQueen, J. B.: Some methods for classification and analysis of multivariate observations. 5th Berkeley Symp. On Math. Stat. and Prob., pp. 281-297 (1967)
- Marcus, A. and Maletic, J. I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. 25th ICSE'03, pp. 125-135 (2003)
- Marcus, A., Xie, X., and Poshyvanyk, D.: When and how to visualize traceability links? TEFSE 2005, Nov. 8, California, USA, pp. 56-61 (2005)
- Merten, T., Juppner, D., and Delater, A.: Improved representation of traceability links in requirements engineering knowledge using Sunburst and Netmap visualization. 4th MARK, Trento, pp. 17-21 (2011)
- Nakagawa, H., Matsui, S. and Tsuchiya, T.: A visualization of specification coverage based on document similarity. IEEE/ACM 39th Int. Conf. on Software Engineering Companion (ICSE-C), pp. 136-138 (2017)
- Nishikawa, K., Washizaki, H., Oshima, K., and Mibe, R.: Recovering transitive traceability links among software artifacts. ICSME 2015, Bremen, Germany, pp. 576-580 (2015)
- Penta, M. D., Gradara, S., and Antoniol, G.: Traceability recovery in RAD software systems. IWPC 2002, pp.207-216 (2002)
- Pilgrim, J. von, Vanhooft, B., Schulz-Gerlach, I., and Bervers, Y.: Constructing and visualizing transformation chains. 4th ECMDA-FA '08, Heidelberg, pp. 17-32 (2008)
- Prefuse, the prefuse information visualization toolkit, 2011, <http://prefuse.org/>
- Rocco, J.D., Ruscio, D.D., Iovino, L., and Pierantonio, A.: Traceability visualization in metamodel change impact detection. GMLD'13, Montpellier, France, pp. 51-62 (2013)
- Rodrigues, A., Lencastre, M., Filho, G. A. de A. C.: Multi-VisioTrace: traceability visualization tool. 10th Inter. Conf. on the Quality of Information and Communication Technology, pp. 61-66 (2016)
- Roman, G. C. and Cox, K. C.: Program visualization: the art of mapping programs to picture. Proc. Of Int. Con. on Software Engineering, pp. 412-420 (1992)
- Settimi, R., Cleland-Huang, J., Ben Khadra, O., Mody, J., Lukasik, W., and DePalma, C.: Supporting software evolution through dynamically retrieving traces to UML artifacts. 7th IWPSSE, Kyoto, Japan, pp. 49-54 (2004)
- Shneiderman, B.: Tree visualization with tree-maps: 2d space-filling approach. ACM Transactions on Graphics (TOG), 11(1), pp. 92-99 (1992)
- Singhal, A.: Modern Information Retrieval: a brief overview. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol. 24, No. 4, pp. 35-42 (2001)
- Spanoudakis, G. and Zisman, A.: Software traceability: a roadmap. Advances in Software Engineering and Knowledge Engineering, Vol. 3: Recent Advances, (ed) S.K Chang, World Scientific Publishing (2005)
- Terrier IR platform, 2010, extracted from <http://terrier.org/>
- Van Amstel, M. F., van den Brand, M.G.J. and Serebrenik, A.: Traceability visualization in model transformations with TraceVis*. ICMT 2012, LNCS 7307, pp. 152-159 (2012)
- van Ravensteijn, W.J.P.: Visual traceability across dynamic ordered hierarchies. Master's thesis, Eindhoven University of Technology (August 2011)
- Voytek, J. B. and Nunez, J. L.: Visualizing non-functional traces in student projects in information systems and service design. CHI 2011, Vancouver, Canada (2011)

- Wang, X., Lai, G., and Liu, C.: Recovering relationships between documentation and source code based on the characteristics of software engineering. *Electronic Notes in Theoretical Computer Science* 243, Elsevier B. V., pp. 121-137 (2009)
- Watkins, R. and Neal, M.: Why and how of requirements tracing. 5th ASM, California, pp.104-106 (1994)
- Witten, I. H., Paynter, G. W., Frank, E., Gutwin, C., and Nevill-Manning, C. G.: Kea: practical automatic keyphrase extraction. 4th ACM DL, Berkeley, pp. 254-255 (1999)
- Woodruff, A. and Stonebraker, M.: VIDA: visual information density adjuster. Tech. Report CSD-97-968, Univ. of California, Berkeley, Calif., Sep. (1997)
- Zhou, X., Huo, Z., Huang, Y., and Xu, J.: Facilitating software traceability understanding with ENVISION. COMPSAC 2008, pp. 295-302 (2008)

5.5 An End-to-End Model-based Approach to Support Big Data Analytics Development

Khalajzadeh, H., Simmons, A., Abdelrazek, M., **Grundy, J.C.**, Hosking, J.G., He, Q., An End-to-End Model-based Approach to Support Big Data Analytics Development, *Journal of Computer Languages*, Volume 58, June 2020, Elsevier

DOI: [10.1016/j.cola.2020.100964](https://doi.org/10.1016/j.cola.2020.100964)

Abstract: We present BiDaML 2.0, an integrated suite of visual languages and supporting tool to help multidisciplinary teams with the design of big data analytics solutions. BiDaML tool support provides a platform for efficiently producing BiDaML diagrams and facilitating their design, creation, report and code generation. We evaluated BiDaML using two types of evaluations, a theoretical analysis using the “physics of notations”, and an empirical study with 1) a group of 12 target end-users and 2) five individual end-users. Participants mostly agreed that BiDaML was straightforward to understand/learn, and prefer BiDaML for supporting complex data analytics solution modeling than other modeling languages.

My contribution: Developed initial ideas for this research, co-designed approach, co-supervised the two post-doctoral fellows, co-authored significant parts of paper, lead investigator for funding for this project from ARC

An End-to-End Model-based Approach to Support Big Data Analytics Development

Hourieh Khalajzadeh¹, Andrew Simmons², Mohamed Abdelrazek², John Grundy¹, John Hosking³,
Qiang He⁴

¹Faculty of Information Technology, Monash University, Australia

²School of Information Technology, Deakin University, Australia

³Faculty of Science, University of Auckland, New Zealand

⁴School of Software and Electrical Engineering, Swinburne University of Technology, Australia

{hourieh.khalajzadeh, john.grundy}@monash.edu, {a.simmons, mohamed.abdelrazek}@deakin.edu.au,
j.hosking@auckland.ac.nz, qhe@swin.edu.au

Abstract—We present BiDaML, an integrated suite of visual languages and supporting tool to help multidisciplinary teams with the design of big data analytics solutions. BiDaML tool support provides a platform for efficiently producing BiDaML diagrams and facilitating their design, creation, report and code generation. We evaluate BiDaML using two types of evaluations, a theoretical analysis using the “physics of notations”, and an empirical study with 1) a group of 12 target end-users and 2) five individual end-users. Participants mostly agreed that BiDaML was straightforward to understand/learn, and prefer BiDaML for supporting complex data analytics solution modeling than other modeling languages.

Keywords—big data analytics, big data modeling, big data toolkits, domain-specific visual languages, multidisciplinary teams, end-user tools

1. INTRODUCTION

Using big data analytics to improve decision-making has become a highly active research and practice area [1, 2]. Gartner’s technical professional advice [3] recommends six stages for machine learning applications development: classifying the problem, acquiring data, processing data, modeling the problem, validation and execution, and finally deployment. Traditionally, advanced machine learning knowledge and experience of complex data science toolsets were required for data analytics applications. Emerging analytics approaches seek to automate many of these steps in model building and its application, making machine learning technology more accessible to those who lack deep quantitative analysis and tool building skills [4].

Recently, a number of data analytics and machine learning tools have become popular, providing packaged data sourcing, integration, analysis and visualization toolkits oriented towards end-users. These include Azure ML Studio [5], Amazon AWS ML [6], Google Cloud ML [7], and BigML [8], as reviewed in [9]. Many of these tools do not require programming language knowledge and are based on simple drag-and-drop interfaces. However, they mostly focus on the machine learning algorithms and sometimes one-click deployment, but lack domain knowledge and business problem capture, modeling, traceability to the solution and validation of the solution against the problem. They also lack an explanation of the model from an end-user perspective.

Data analytics and machine learning steps need to be more tightly connected to the control and management of business and requirements engineering processes [10]. However, the primary focus of most current big data analytics tools and technologies is on storage, processing, and in particular data analysis and machine learning tasks. Current data analytics tools rarely focus on the improvement of end-to-end processes [11]. To address this issue, better integration of data science, data technology, and process science is in demand [11]. Data science approaches tend to be process-agonistic whereas process science approaches tend to be model-driven without considering the evidence hidden in the data. Scalable process mining analytics need to be brought to big data toolkits while enabling them to be easily tailored to accommodate domain-specific requirements [11]. Tools filling these gaps would be very useful for multidisciplinary teams where a range of complicated steps and users with different skillsets are involved. E.g. to empower domain experts and business managers during the discovery and exploration phase with the tools to model the problem, extract insights/patterns, and design predictive/clustering models (if feasible) before they involve other stakeholders such as data scientists and software engineers.

We present our novel approach to addressing this problem - Big Data Analytics Modeling Languages (BiDaML). BiDaML, extended from [10], is a set of domain-specific visual models at differing levels of abstraction, to capture and refine requirements and specify different parts of the data analytics process. Through these domain-specific visual languages (DSVLs), we aim to make data analytics design more accessible to multidisciplinary teams and facilitate dialogues with expert data scientists and software engineers. BiDaML provides better tool

support and collaboration between different users while improving the speed of implementing data analytics solutions. This work is extended from [10] mainly in three ways: 1) The diagrams and notations are extended and are significantly different in this new version, i.e., the number and type of diagrams, number and type of notations, the generated outputs, the evaluations, etc. 2) The Physics of Notations (PoN) evaluation done in [10] is significantly different from the comprehensive PoN conducted for this paper that led to essential changes to the notations. 3) A completely new user study with a group of target end-users including data scientists and software engineers was conducted and the results were analyzed and included in the extended version. In the group study, the notations and models are compared with the existing modeling frameworks. We follow two objectives in this paper: 1) The major objective is to present domain specific modeling languages and a support tool for designing, documenting, and communicating data analytics solutions. 2) The secondary objective that has not been elaborated comprehensively in this paper is generating code, recommending solutions, and reusing the existing solutions.

The rest of the paper is organized as follows. In Section 2, the background and motivation of this research are described with a real-world example of a property price prediction system development. Our approach is discussed in Section 3 and evaluated in Section 4. Limitations and future directions are discussed in Section 5. A comprehensive comparison to related work is presented in Section 6. Finally, we draw conclusions and discuss key future directions in Section 7. BiDaML tool, data and evaluation results can be accessed publicly from [18].

2. MOTIVATING EXAMPLE

In this section, we discuss data analytics steps as well as different types of communication between users in a data analytics project. A real property price prediction example is then demonstrated to reflect the issues and some key challenges in the process of data analytics.

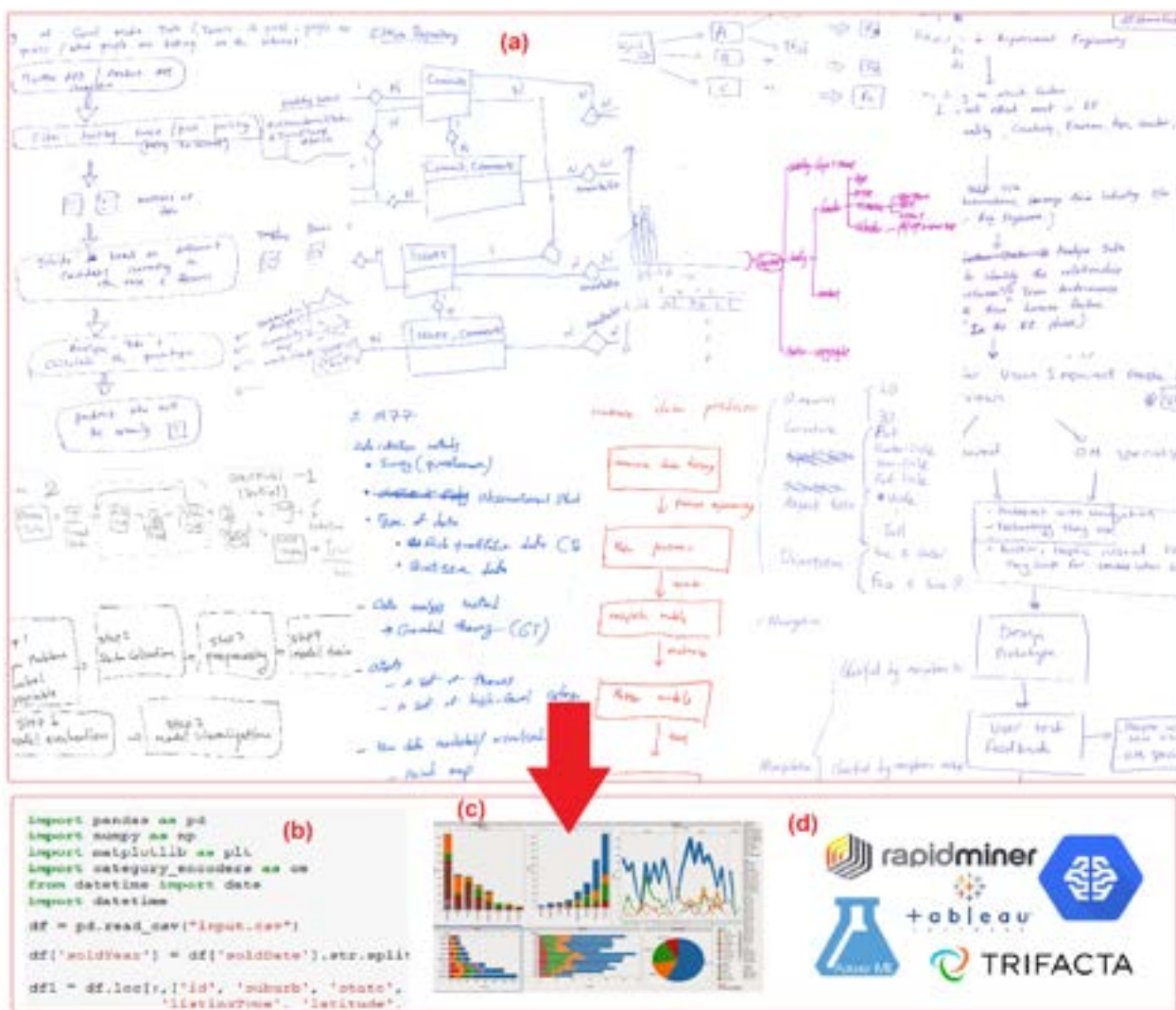


Figure 1. How data scientists currently design their big data analytics solutions. Examples (top) captured during our group user study.

2.1 Data Analytics Process Steps

The key steps that data scientists take to design their solutions are illustrated in Figure 1. Business owners, business analysts, data analysts and data scientists need to conduct meetings and interviews with domain experts

and users and therefore, generate many handwritten/ad-hoc notations and documentations and meeting notes. Then, they acquire required datasets from different resources, obtain access to government information, integrate all datasets with different formats by communicating with each other to discuss the analyses (b) writing scripts, (c) analyzing and visualizing data, and (d) finally use different tools to design their approaches and develop their models. As shown in Figure 1(a), based on the data we collected during our group user study, there is, in fact, no unified language for these stakeholders (including non-technical domain experts and business managers through to technical data scientists and software engineers with mostly no domain knowledge) to facilitate communication throughout the project and also no unified tool to record, capture and document all the steps involved. Therefore, the higher-level stakeholders need to wait for the technical team to derive usable and deployable models to obtain updates on their progress. In this section, we show an example of a property price prediction system development to discuss some of the problems the users involved in the project face during the solution design process.

2.2 Example: Property Price Prediction

Consider a situation where data analytics is intended to be used for predicting property prices, as shown in Figure 2. In this scenario, a real estate agency wants to improve agents' focus and outcomes by looking for patterns in a large amount of real estate, government and financial data. In order to solve this problem using conventional approaches, such end-users in multidisciplinary teams would need to have a basic knowledge of data analytics programming languages such as Python and R and also a basic knowledge of data science. Due to the lack of necessary knowledge, the company employs a technical team of software engineers and data scientists to work on the problem. The user needs to upload a dataset, choose features based on the quality of the features, apply data type casting and data cleansing, and finally filter the features to build the dataset as an input for the prediction model. Then the user chooses the best model, algorithm, and validation method and adapts the characteristics based on problem requirements.

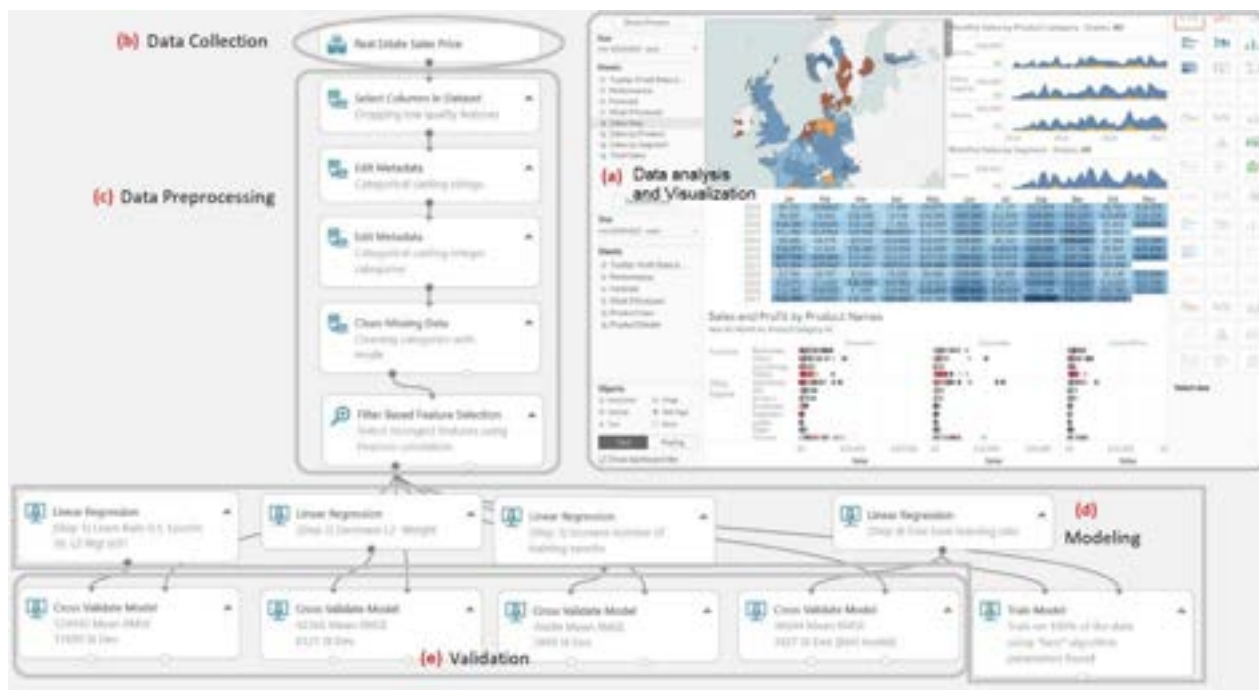


Figure 2. Property price prediction using Tableau and Azure ML Studio

The team decides to use Azure ML Studio to do some initial analyses on the Ames Housing dataset [12] before purchasing data from a data provider. In Figure 2, (a) the user analyses and visualizes data in Tableau, then using ML Azure Studio (b) drags the “Real Estate Sales Price” module to upload input data, (c) adds “Select Columns in Dataset”, “Edit Metadata”, “Clean Missing Data”, and “Filter Based Feature Selection” modules to prepare and clean data, (d) “Linear Regression” and “Train Model” modules to apply linear regression and train the model, and finally (e) “Cross Validate Model” module to validate the model. These steps are low-level machine learning operations and the user needs to have data science knowledge to choose the modules properly and change their parameters. If the user wants to use a pre-processing method or apply a model that does not exist in the list of modules, knowledge of programming languages such as Python and R is required to embed code in order to improve steps and add features.

The new challenge that arises is that the company realizes that the team lacks sufficient understanding of important domain knowledge; therefore, appoints a senior real estate agent, highly experienced with the nuances in the domain, to help the team build the analytics solution. The team struggles with a lack of a common dialect between engineers, scientists and domain experts. Eventually, the team realizes that the solution is not ready due to issues in the available dataset that need to be rectified and it takes a long time for the team to finally design and

build a working analytics solution. A few months later, after the technical team has been disbanded, the company realizes that adding new features, e.g. public transports in the area, or building a new model to predict who would be willing to sell their property in a given suburb might be beneficial. In addition, the company observes degradation in the performance of the existing property price prediction model, which needs to be updated. The company decides to recruit a new team to apply the new capability, however, they struggle to reuse and update the existing solution due to lack of business knowledge and lack of clear documentation or an integrated framework to allow them to reuse or integrate models.

2.3 Key Challenges

As illustrated above in Figure 2, there is no traceability back to the business needs/requirements that triggered the project. Furthermore, communicating and reusing existing big data analytics information and models is a challenge for many companies new to data analytics. Users need to be able to collaborate with each other through different views and aspects of the problem and possible solutions. Current practices and tools do not cover most activities of data analytics analysis and design, especially the critical business requirements. Most current tools focus on low-level data analytics process design, coding and visualization of results and they mostly assume data is in a form amenable to processing. In reality, most data items are in different formats and are not clean, integrated or usable by the machine learning models and great effort is needed to source the data, integrate, harmonize, pre-process and cleanse it. Moreover, only a few of the machine learning tools offer the ability for the data science expert to embed new code and expand algorithms and provide visualizations for their needs.

Data processing and machine learning tasks are only a small component in the building blocks necessary to build real-world deployable data analytics systems [13]. As Figure 3 illustrates, these tasks cover a small part of data and machine learning operations and model deployment. At the same time, conventional business and management modeling tools do not usually support many key data analytics steps including data pre-processing and machine learning steps. There is a need to capture the high-level goals and requirements for different users such as domain expert, business analyst, data analyst, data scientist, software engineer, and customers and relate them to low-level steps and capture details such as different tasks for different users, requirements, objectives, etc. Finally, embedding code and changing features based on the users' requirements require data science and programming knowledge.



Figure 3. Data analytics steps (adapted from [13])

3 OUR APPROACH

As the example in Section 2 shows, many current big data analytics tools provide only low-level data science solution design, despite many other steps being involved in solution development. Therefore, a high-level presentation of the steps to capture, represent and communicate the business requirements analysis and design, data pre-processing, high-level data analysis process, solution deployment, and data visualization is required.

3.1 BiDaML Visual Language

We present BiDaML, a set of domain-specific visual languages using five diagram types at different levels of abstraction to support key aspects of big data analytics. These five diagram types cover the whole data analytics software development life cycle from higher-level requirement analysis and problem definition through the low-level deployment of the final product. These five diagrammatic types are:

- **Brainstorming diagram** which provides an overview of a data analytics project and all the tasks and sub-tasks that are involved in designing the solution at a very high level. Users can include comments and extra information for the other stakeholders;
- **Process diagram** which specifies the analytics processes/steps including key details related to the participants (individuals and organizations), operations, and conditions in a data analytics project capturing details from a high-level to a lower-level;
- **Technique diagrams** which show the step by step procedures and processes for each sub-task in the brainstorming and process diagrams at a low level of abstraction. They show what techniques have been used or are planned to be used and whether they were successful or there were any issues;

- **Data diagrams** which document the data and artifacts that are produced in each of the above diagrams at a low level, i.e. the technical AI-based layer. They also define in detail the outputs associated with different tasks, e.g. output information, reports, results, visualizations, and outcomes;
- **Deployment diagram** which depicts the run-time configuration, i.e. the system hardware, the software installed on it, and the middleware connecting different machines to each other for development related tasks.

Figure 4 shows an overview of the BiDaML approach on the left side, and how these diagrams are created and connected from high level to low level for the Property Price Problem, in the right side. Different users i.e., domain experts, business owners, business analysts, data analysts/scientists, and software engineers create and modify the diagrams in a collaborative way via the BiDaML tool. In the property price prediction example, a brainstorming diagram is defined for the project to assist all the stakeholders to communicate and specify the high-level objectives, targets, tasks, etc. Then, at a lower level to include more details and specify the organizations and participants, we use a process diagram, which is also a unique diagram for each project. Every task in brainstorming and process diagrams can be further extended by technique and data diagrams at different levels. As many technique and data diagrams as is necessary can be defined by data scientists and software engineers to communicate their findings and progress with the other members of the team in a more communicable language than the informality of the notes of Figure 1. Finally, a deployment diagram, defined for development related tasks in the data analytics problem, models the deployment related details at a low level. A deployment diagram is unique for each project, and reuses the notations and elements defined in the previous diagrams to communicate the deployment strategies used by software engineers. With the users having defined this set of diagrams, the tool can then generate and inform users of reports, documentations, Python code, and API access code.

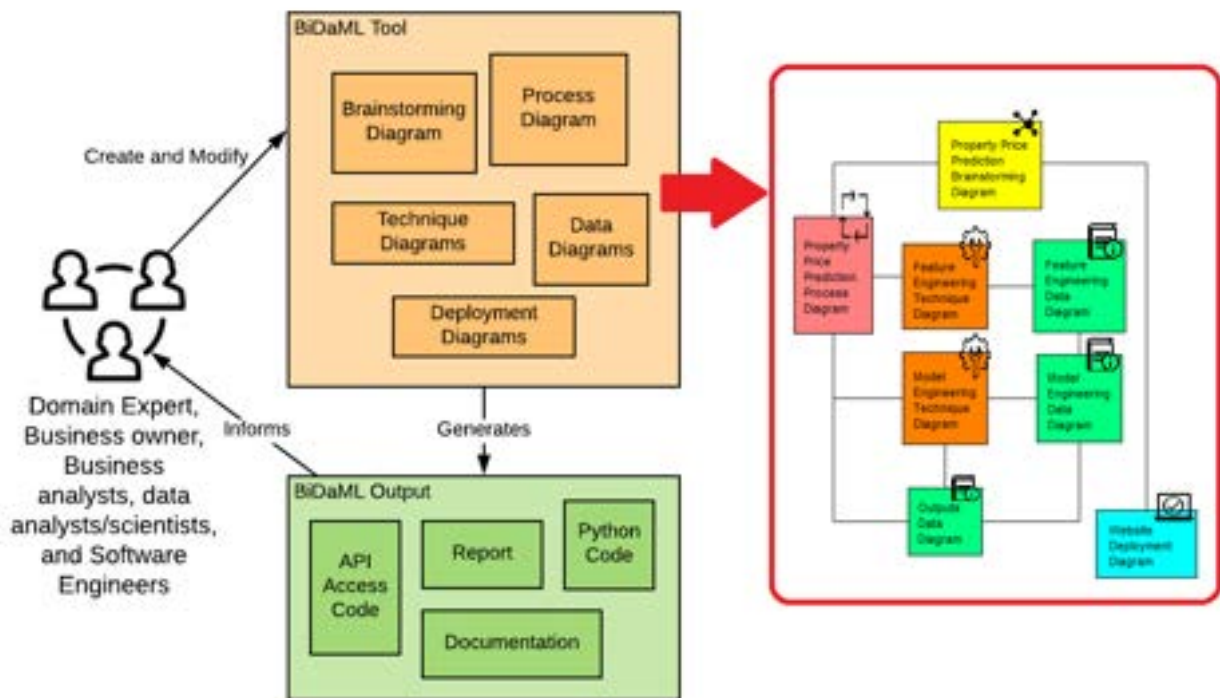


Figure 4. BiDaML overview (left), and overview of the diagrams created for the property price problem (right).

3.1.1 Brainstorming Diagram

A data analytics brainstorming diagram covers the entirety of a data analytics project expressed at a high-level. There are no rules as to how abstractly or explicitly a context is expanded. The diagram overviews a data analytics project in terms of the specific problem it is associated with, and the tasks and subtasks to solve the specific problem. It supports interactive brainstorming and collaboration between interdisciplinary team members to identify key aspects of a data analytics project such as its requirements implications, analytical methodologies, and specific tasks.

Figure 5 shows the visual notation we use for a brainstorming diagram. It comprises a red hexagon icon representing the data analytics problem and yellow ellipses to show to-do tasks with which the problem is associated. The red hexagon is intended to emphasize the problem, e.g. “Property Price Prediction” in our example, while the yellow ellipses provide a notational association with Post-it notes. High-level tasks connected directly to the problem are automatically changed to bright orange post-it notes with fixed pins to highlight their importance. For our motivating example, “price prediction feature engineering” is one of the high-level tasks which is further

broken down to “clean data”, “impute missing value”, “add new features”, “choose features”, etc, as lower level to-do tasks. Specific information to share between the members is captured as comment icons. Finally, input icons are used in case specific data items need to be connected to any of the tasks.

We characterize the building blocks of a data analytics software system into four groups: Domain and business-related activities (BusinessOps); data-related activities (DataOps); artificial intelligence and ML-related activities (AIOps); and development and deployment activities (DevOps). BusinessOps covers domain and business knowledge and requirement gathering, modeling, and analysis. DataOps includes data collection/ingestion, data validation cleansing, wrangling, filtering, union, merge, etc. AIOps covers feature engineering and model selection, model training and tuning. Finally, DevOps covers model integration and deployment, monitoring and serving infrastructure. Blue dashed ellipses (operation separators) are used to group these operations in order to make it easier for the different types of stakeholders to focus on their specific aspects of the problem and to see how this fits in with the overall system. An undirected line is then used to connect the problem to the tasks and tasks to the other tasks, comments, and inputs.



Figure 5. Brainstorming diagram notational elements.

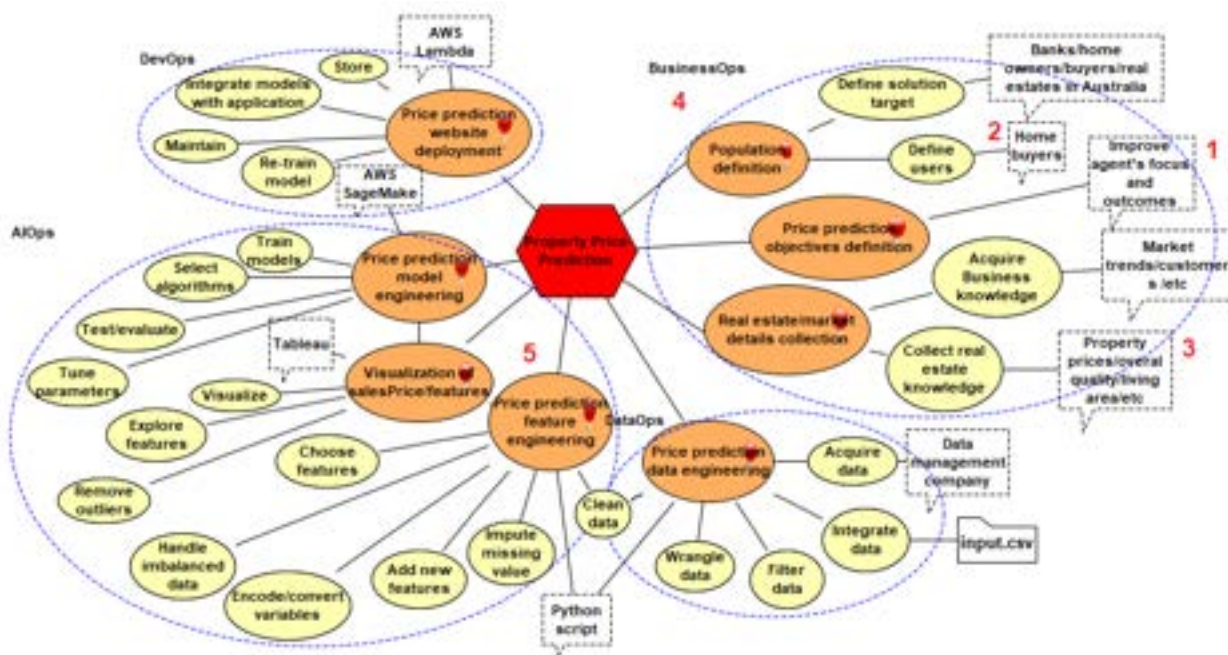


Figure 6. A brainstorming diagram example for the property price problem

Figure 6 depicts a high-level brainstorming diagram for our property price prediction example from Section 2. As some insights, from this figure we can see that:

1. The objective of the data analytics problem is to improve the agent’s focus and outcomes;
2. The project’s target population is home buyers;
3. Its implications are the identification of relationships between the sales price and features such as overall quality, living area, etc;
4. Population definition, price prediction objectives definition, and real estate/market details collection are the high level tasks to be conducted as BusinessOps before acquiring and accessing data; and
5. In order to conduct price prediction feature engineering, clean data, impute missing value, add new features, and choose features, are some of the tasks to complete.

3.1.2 Process Diagram

The key business processes, organizations, and stakeholders involved as well as the project flow in a data analytics application are shown in a process diagram, whose basic notation is shown in Figure 7. We have adapted and simplified the Business Process Modeling Notation (BPMN) [14] to specify big data analytics processes at several levels of abstraction. Process diagrams support business process management, for technical users such as data analysts, data scientists, and software engineers as well as non-technical users such as domain experts, business users and customers, by providing a notation that is intuitive to business users, yet able to represent complex process semantics.

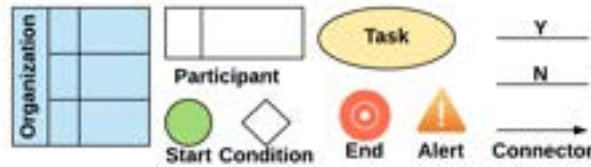


Figure 7. Process diagram notational elements.

In this diagram type, we use different “pools” for different organizations and different “swim lanes” for the people involved in the process within the same organization. For example, “Business analyst/owner”, “Data analyst”, “Data scientist”, and “Software engineer” are the users in “data analytics solution provider”, as one of the organizations involved in our motivating example. To facilitate cognitive integration of the different BiDaML diagram types, tasks are reused from the brainstorming diagram and any changes to tasks in one diagram are automatically propagated by the tool to the others. There might be some tasks not being used in the process diagram or some tasks being created in the process diagram that were not initially in the brainstorming diagram. The reason some of the tasks in the brainstorming diagram might not be followed in the process diagram is that they are not related to the other participants or organizations or they are high level tasks and have already been resolved in the previous steps. Also, some new lower-level tasks might be expanded from high level tasks and added to the process diagram that were not initially considered in the high-level brainstorming diagram. A green circle is used to show the start of the project and a target sign to show the ending point of the project. For instance, a data analytics project starts for a real estate manager when they have some data to be analyzed, and it finishes when users get predictions from the website and follow up with the business manager. Tasks, conditions, and alerts trigger other events and redirect the process to the other users in the same or different pool. Diamonds show different decision points that can adjust the path based on conditions, and the unexpected events that can change the process at any step. Lines with yes (Y) and no (N) signs are used to connect from conditions to the appropriate tasks and directed arrows are used to connect the tasks to other tasks, conditions, alerts and the ending point based on the order and priorities. Except from the task notation that is re-used from the brainstorming diagram, the other process diagram notations reuse symbols and syntax from BPMN where possible, to let users who are familiar with BPMN symbols and syntax to more easily remember and re-use the symbols.

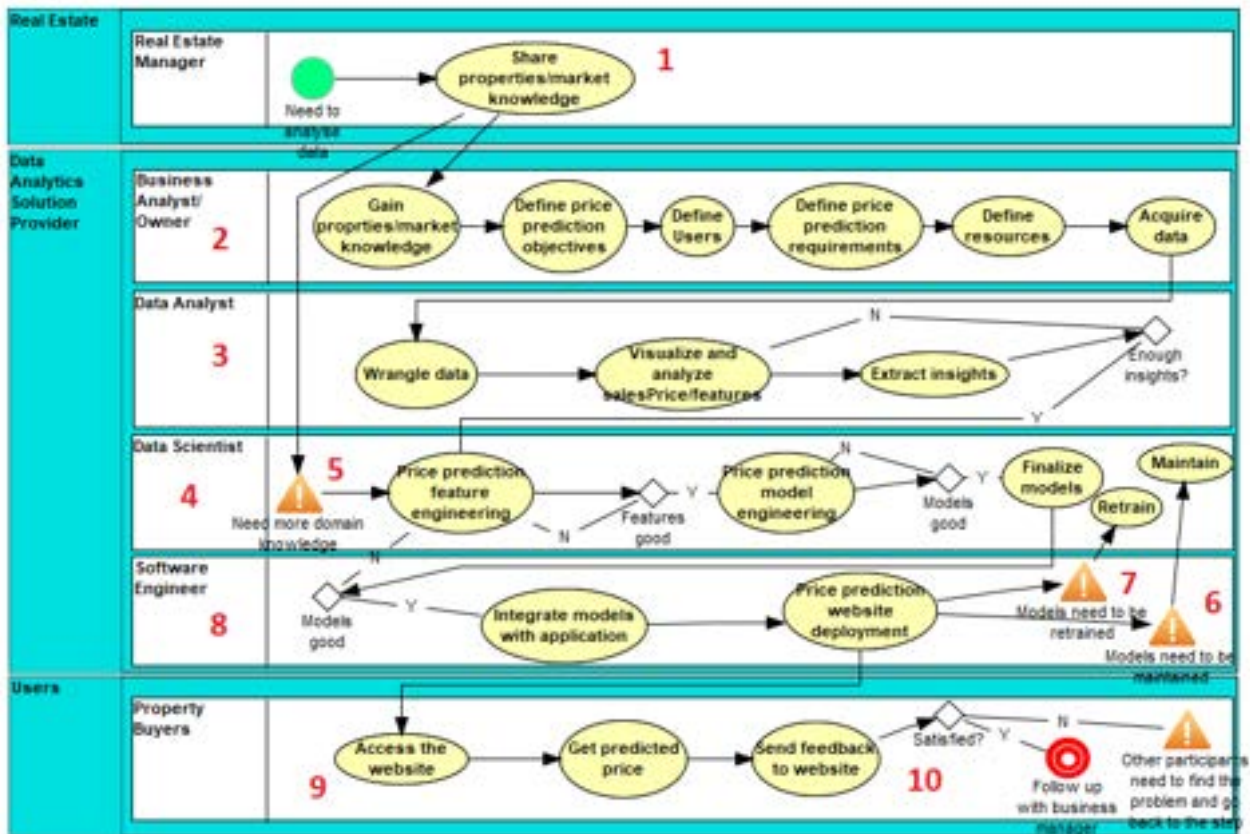


Figure 8. A process diagram example for the property price problem

A high-level process diagram for our property price prediction example is shown in Figure 8. In this, 1) a real estate manager from a real estate organization shares background knowledge with the business manager/analyst. 2) The business analyst /owner then defines the price prediction objectives, requirements, users, etc and acquires data from the related companies and organizations based on the budget and requirements and passes the dataset along with the domain knowledge to the data analysts/scientists team. 3) A data analyst applies DataOps, extracts insights, and communicates the insights with the 4) data scientists to allow them to apply AIOps and share the finalized models with software engineers. These collaborations between domain expert, business manager/analysts, data analysts/scientists, and software engineers do not happen only once; they might 5) return to previous steps to collect more domain knowledge and insights based on an unexpected event, 6) maintain and 7) retrain models, in case the condition is not fulfilled. Once models are finalized, 8) software engineers can deploy the models and 9) users can access the property price prediction website to send requests. Participants need to be involved later to retrain, maintain, and improve the models based on the 10) users' feedback. In this case, an alert needs to be triggered and the business managers must find the responsible participant to improve the process and consequently the outcome.

3.1.3 Technique Diagram

Data analytics technique diagrams extend the brainstorming diagram to low-level details specific to different big data analytics tasks. For every task, the process is broken down to the specific stages and the technique used to solve a specific task is specified. Figure 9 shows the technique diagram notation. Tasks and alert symbols are reused from the brainstorming and process diagrams; a green hexagon sign is used to specify the techniques used and a tick sign to specify that a technique leads to successful results while alerts rectify the issues/challenges faced that need to be taken into account. The green color is associated with the solution and the reason for using a hexagon as the shape is to include a variety of different shapes based on the physics of notations recommendations. Tick and alert signs are automatically associated with success and issues in the users' minds with no need to further remember new signs. Directed arrows are used to connect tasks to the techniques used/planned to be used for them, and the techniques to the other techniques and outcomes.



Figure 9. Technique diagram notational elements

Two different technique diagrams for data wrangling and price prediction model engineering are shown in Figure 10. In Figure 10 (a), (a1) sold properties data needs to be purchased from a data provider, and some additional datasets collected from government websites, or (a2) Python pandas library can be used for parsing input file, data reshaping, and data integration, however, it needs basic data science and programming knowledge. Moreover, in Figure 10 (b), (b1) XGBoost and (b2) Lasso regression algorithms are used to train initial models and finally (b3) the average of the predictions from these two models is used as the final prediction. A (b4) cross-validation technique is used for the test/evaluate task. We can create such diagrams for every task and sub-task in brainstorming and process diagrams.



Figure 10. Technique diagram examples for a) data wrangling and b) price prediction model engineering tasks in the property price problem

3.1.4 Data Diagram

To document the data and artifacts consumed and produced in different phases described by each of the above diagrams, a low-level data diagram is presented, using the notation in Figure 11. Data diagrams support the design of data and artifacts collection processes. They represent the structured and semi-structured data items involved in the data analytics project in different steps and the data items, reports, models and parameters generated throughout the project. Keeping track of data helps the explainability of black-box machine learning models. A high-level data diagram can be represented by connecting the low-level diagrams for different BusinessOps, DataOps, AIOps, and DevOps.

In Figure 11, tasks are reused from brainstorming and process diagrams and new notations are defined for different types of data and artifacts, e.g. clipboards to represent reports and dashed grey rectangles to represent information. Data items are shown by green file types to reflect excel/datasets and blue file types to reflect source code/models. Directed arrows are used to connect tasks to the data items and artifacts and vice versa.



Figure 11. Data diagram notational elements.

Three different data diagrams for “price prediction feature engineering”, “property price model engineering” and “integrate models with applications” tasks for our price prediction problem are created and shown in Figure 12. Here, data and artifacts related to all tasks in brainstorming and process diagrams are connected to different data entities. In this case, different data items, features, outliers, the algorithms used, parameters related to these algorithms, models created based on different algorithms, the evaluation metrics used for the models, and finally, the expected outputs are captured. Data and artifacts produced for all the other tasks can be detailed and depicted with different data diagrams.

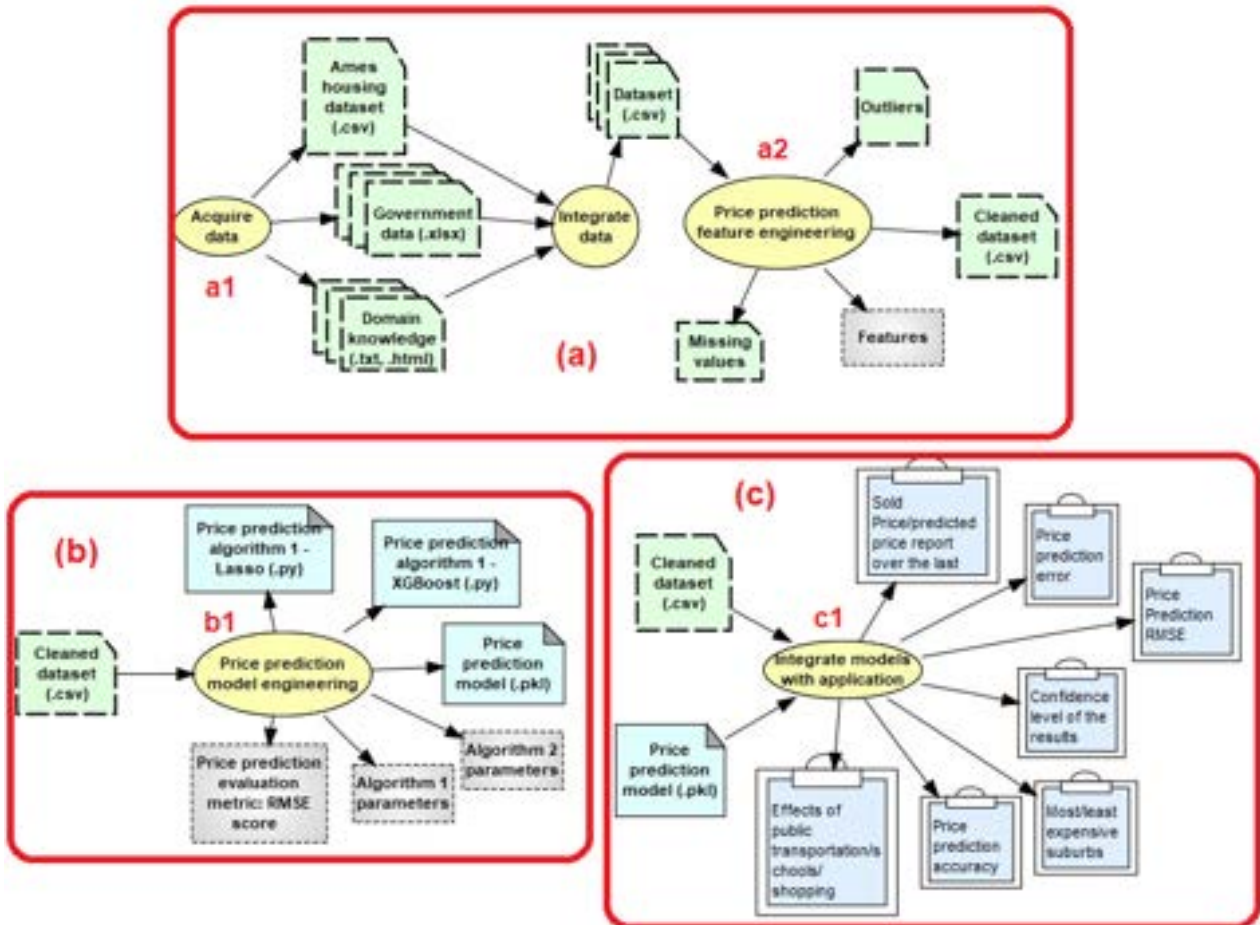


Figure 12. Data diagram examples for a) price prediction feature engineering, b) model engineering and c) integrate models with application tasks in the property price problem

For example, in Figure 12 (a), (a1) Ames housing dataset in CSV format, government data in XLSX format, and domain knowledge in TXT and HTML formats are collected during the “acquire data” task and need to be integrated through the “integrate data” task. Also, (a2) the “price prediction feature engineering” task receives a dataset in CSV format and generates outliers, a cleaned dataset, features, and missing values. In Figure 12 (b), the “price prediction model engineering” task receives the cleaned dataset and generates Python codes, models, parameters, and scores. Finally, in Figure 12 (c), we can see (c1) the outputs and reports that can be extracted using current techniques and data items, such as predicted price versus sale price report, and accuracy, as well as analysis of property prices in different areas based on nearby schools, public transport, and shopping centers.

3.1.5 Deployment Diagram

Since the deployment phase follows the same rules as the deployment process in software development, we had initially adopted the deployment diagram concepts from the context of Unified Modeling Language (UML) [15]. However, as big data analysis requires a focus on distributed cloud platforms, services, and frameworks rather than individual nodes/devices, the visual notation itself draws inspiration from system layer diagrams to visualize the technology stack. The BiDaML deployment diagram notation is shown in Figure 13. Most of the notations are reused from the other BiDaML diagrams.

In a deployment diagram, rectangles indicate the software artifacts and the deployment components and an application logo is used to specify the website and application. Undirected lines from one item to another item specify the relationships between elements.



Figure 13. Deployment diagram notational elements.

Figure 14 shows an example deployment diagram for our property price prediction problem. In this diagram we can see that 1) the browser is running on the client system, 2) the sold properties/suburbs reports and prediction price results will be displayed in the browser using HTML5, 3) AWS Lambda is used to respond to website price prediction requests from clients and run the model, 4) government suburbs data is stored in the real estate MySQL DB, 5) model training tasks are running on AWS Spot instances, and so on. Note the use of vertical stacking in BiDaML to denote layers of the technology stack, e.g. AWS Cloud is used to run AWS Spot Instances that support the Model Training environment. This is because visual containment (e.g. nested execution environments in UML deployment diagrams) can lead to diagrams that are hard to read and use when there are many levels of containment involved [16]. Figure 14 also displays tasks directly on the deployment diagram, an optional feature of BiDaML deployment notation designed to help clarify how the infrastructure will support key tasks, e.g. the Model Training environment exists to support the tasks of training/re-training models on demand, and to make it easier to relate the deployment diagram back to the other diagrams.

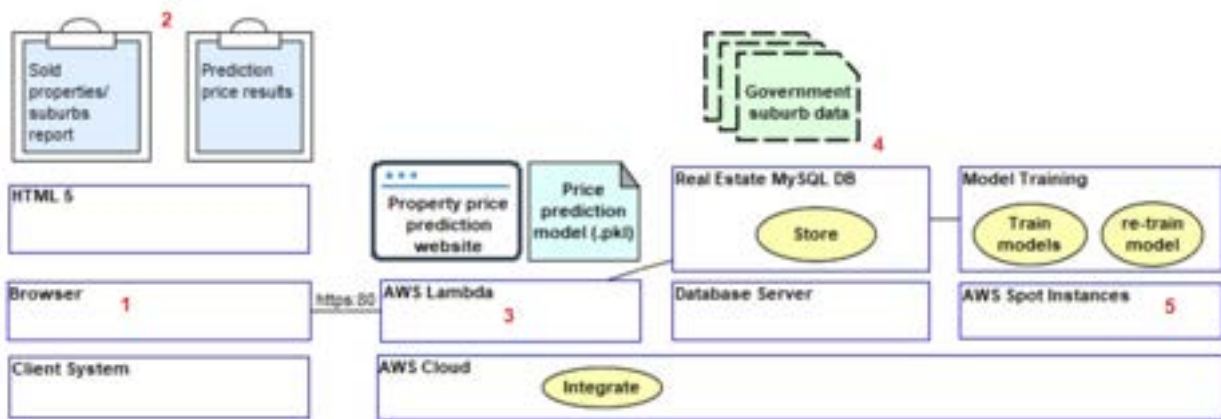


Figure 14. A deployment diagram example for the property price problem utilizing BiDaML deployment notation to show layers of the technology stack and how they support key tasks.

3.2 BiDaML Support Tool

We have developed an integrated design environment for creating BiDaML diagrams. BiDaML tool support aims to provide a platform for efficiently producing BiDaML visual models and to facilitate their creation, display, editing, storage, code generation and integration with other tools. We used MetaEdit+ Workbench [17] to implement our tool. Using MetaEdit+, we created the objects and relationships defined as the notational elements for all of the diagrams, different rules on how to connect the objects using the relationships, and finally how to define low-level sub-graphs for the high-level diagrams. Figure 15 shows (a) how the objects, relationships, and roles are defined, (b) how the rules for connecting objects through relationships are defined, and (c) how the sub-graphs are connected to different objects of a graph, for the overview diagram. These are defined in a similar way for all other BiDaML diagrams.

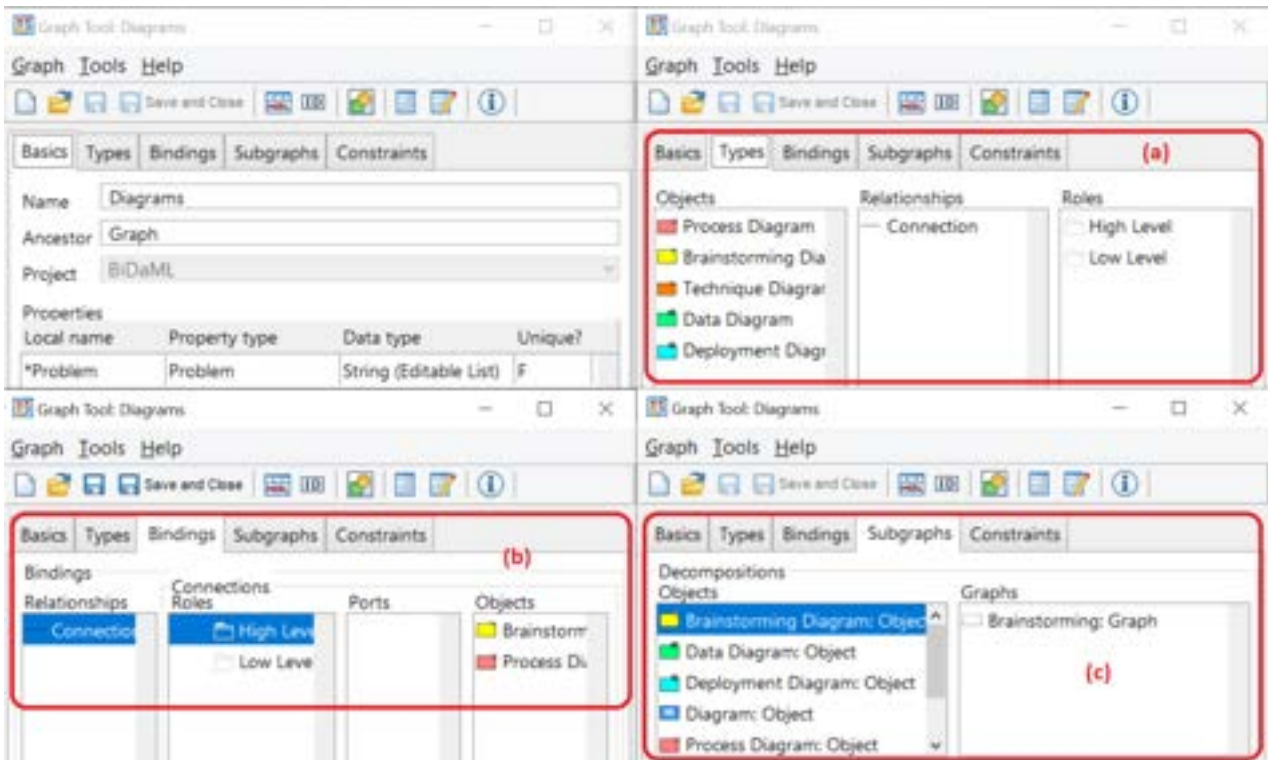


Figure 15. Defining notational elements in MetaEdit+

Once the notational elements are defined and bound and different diagrams are connected together, users can create different diagrams. Figure 16 shows our tool used to create an overview of all the diagrams for the price prediction example we discussed throughout this paper. Starting from an overview of all the diagrams, figure 16 shows that (a) notations can be dragged and dropped to the drawing area and connected together, then (b) different diagrams can be created and connected to each of the notations, and (c) accessed later. Once all the diagrams are created and connected to the appropriate notations, users can (d) generate different types of documentation and reports from the overview diagrams.

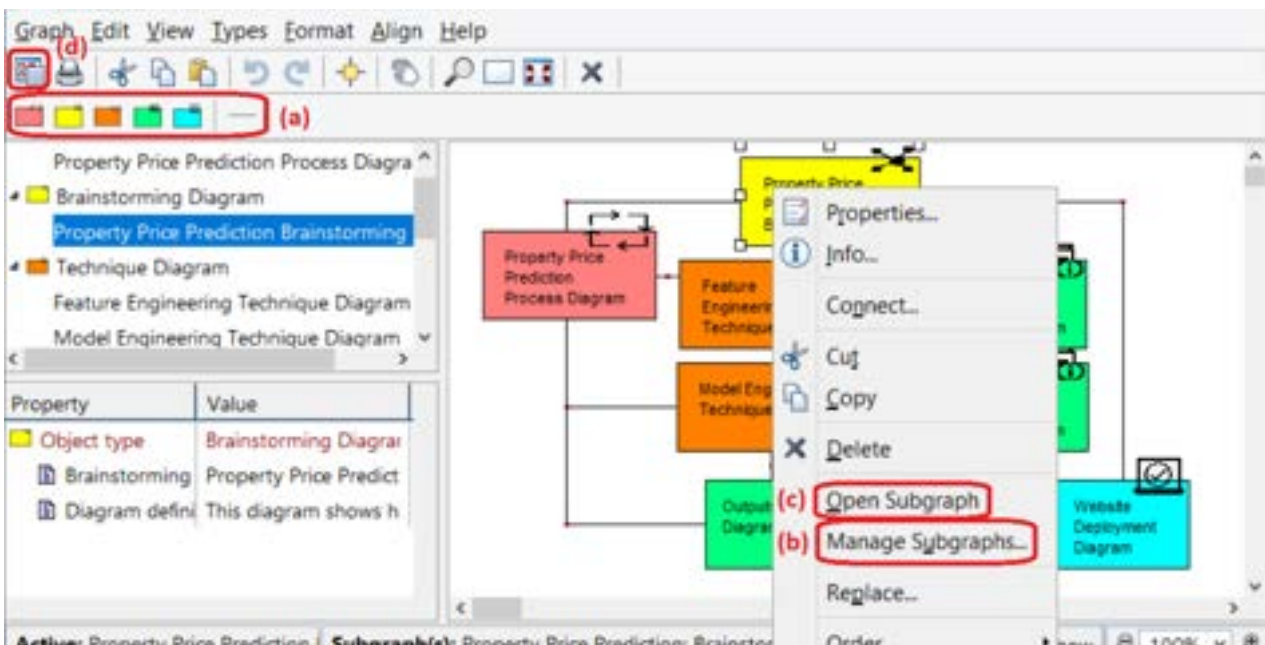


Figure 16. How to create an overview of BiDaML diagrams, connect them together and generate reports

Figure 17 shows the brainstorming created for the same problem, as a sub-graph of the brainstorming diagram notation in the overview diagram. Here, users (a) choose the notations of objects/relationships and (b) modify the properties of the object/relationship. Notations added to the diagram are all listed (c) and details are shown by clicking on the notations (d). Users can click on any of the objects and create a sub-graph i.e., data and technique diagrams for them. Finally, once completed, (e) code generation features can be embedded and modified and (f) finally Python code, BigML API recommendations and reports can be generated for our property price prediction

example in this designed brainstorming diagram. We will explain these in more detail in sections 3.2.1 and 3.2.2 below.

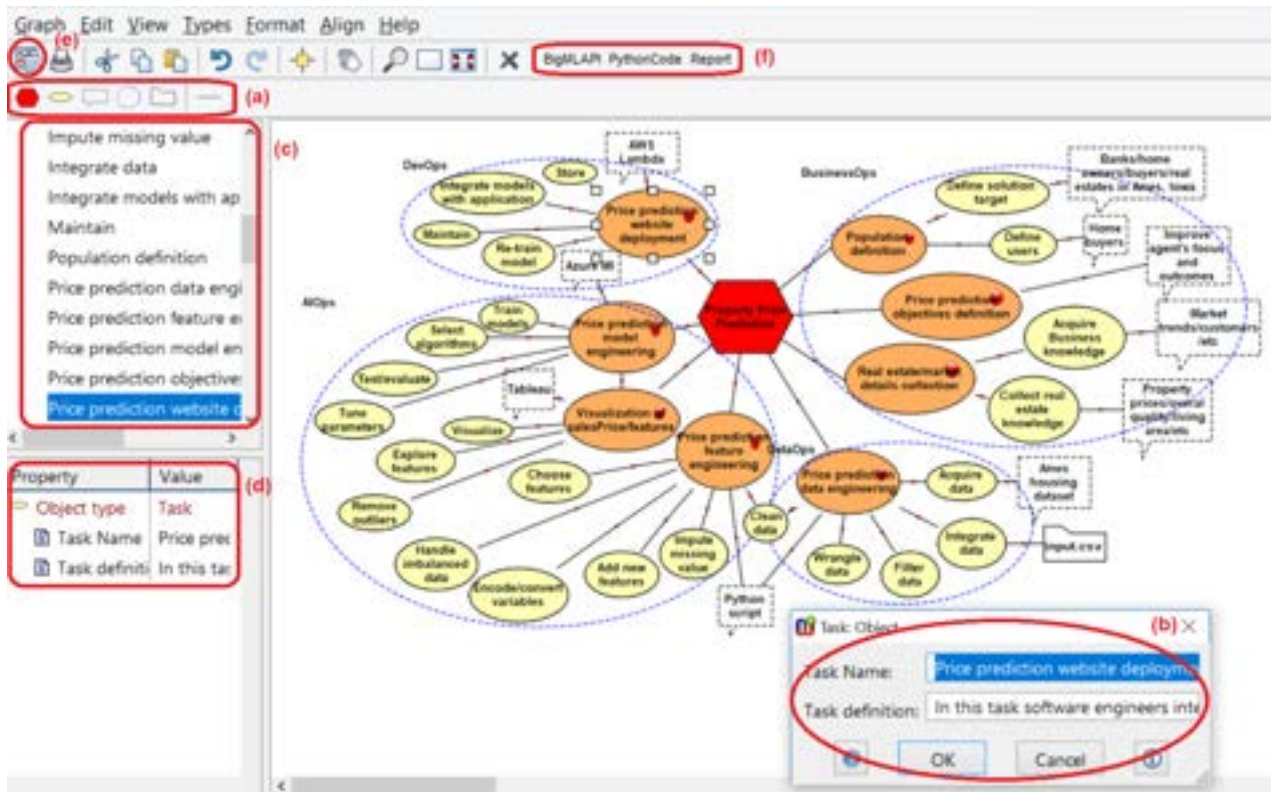


Figure 17. An example of BiDaML tool for creating brainstorming diagram for the property price problem

Figure 18 shows one of the technique diagrams created for the “price prediction model engineering” task in the brainstorming diagram, as a sub-graph of the technique diagram notation in the overview diagram. In this figure, users (a) choose the notations of objects/relationships and create the diagram by dragging, dropping and choosing a name/explanation for the objects. Technique diagrams also have (b) code generation features that can be embedded and modified and (f) reports can also be generated for the technique diagram.

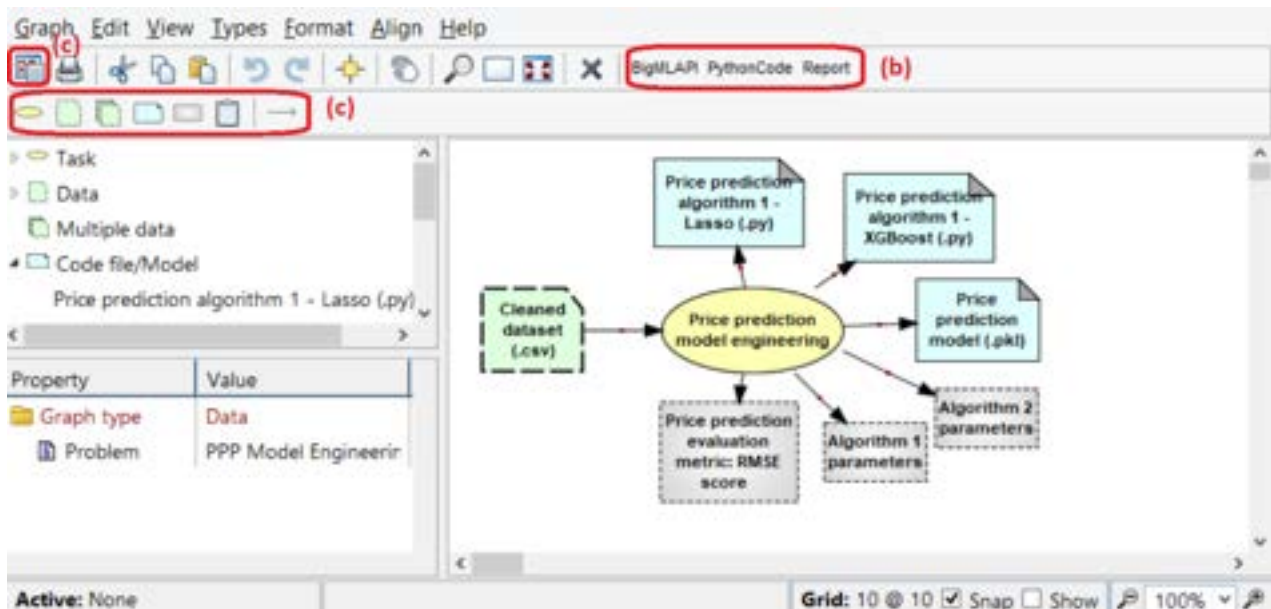


Figure 18. An example of BiDaML tool for creating technique diagram for the property price problem

Once all the diagrams are created and connected, users can obtain outputs and share them with other stakeholders. There are currently two sets of outputs generated from the diagrams. First, a hierarchy of the graphs can be exported to Word and HTML from any of the diagrams. However, since all the sub-graphs are connected together in the overview diagram, the most comprehensive report can be generated and exported to Word/HTML

through the overview diagram. The second set of outputs are Python code/BigML API, and reports that are embedded in the tool and can be traced back.

3.2.1 BiDaML Report Generation

Users can generate a hierarchical report from the overview diagrams, which covers all the high level to low level diagrams from brainstorming to deployment diagram, and also includes any optional metadata associated with objects in the diagrams. For example, task objects have an optional “task definition” field that can be included in the Word document without being presented in the diagrams. This will help to automatically aggregate all the diagrams and information, generate detailed reports and share them with the other stakeholders involved in the project. Some of the reports generated from different industrial projects can be found in [18].

3.2.2 BiDaML Code Generation

For the code generation part, we have currently embedded the common data analytics tasks including “importing libraries”, “read/write files”, “impute missing value”, “visualize”, “wrangle data”, etc and well-known machine learning algorithms such as “XGBoost”, “SVM”, and “linear regression”. The tool iterates through the tasks and techniques and if any of the existing methods are planned for the project as a task or technique, it includes the code in the generated Python file. Then users can modify the source code and work on it instead of writing the code from scratch. Also, users can embed their final code in the tool to reuse for future projects or retraining the existing models. Python code can be generated from brainstorming and technique diagrams. If generated from the brainstorming diagram, the code is more comprehensive since it iterates through the techniques associated with any of the tasks. Regarding API recommendation, we have currently included BigML API recommendation which helps users create their source code, dataset, model, and predictions more easily in order to get started with BigML.io. Setting up the environment variable, i.e., BIGML_AUTH, storing username and API key, etc are all included in the BigML API source code. This can be extended to any other tools and users can add their own API accesses. Figure 19 shows a snippet of the Python code generated from the brainstorming diagram and the report generated from the process diagram. All the diagrams in a project and the revision histories are stored in a Git repository. Any of the stakeholders can access the tool from their computer, modify different diagrams, commit changes to Git, and then push them (e.g. to GitHub) for the other collaborators to access the latest version.

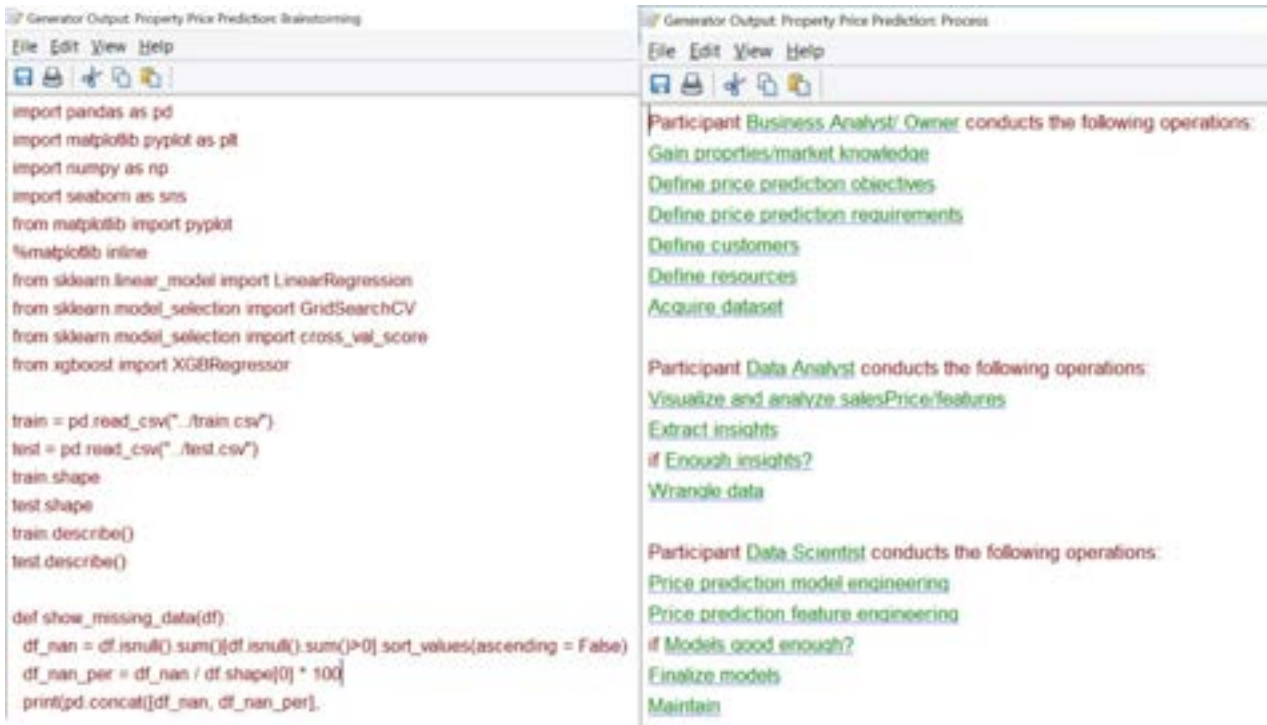


Figure 19. A snippet of the Python code generated from the brainstorming diagram (left) and the report generated from the process diagram (right)

4 EVALUATION

We have evaluated the usability and suitability of BiDaML in two ways. The first was an extensive Physics of Notations evaluation of the visual notation [19]. This allowed a rigorous symbol-by-symbol evaluation of the usability and effectiveness of the visual notation against established theoretical principles without having to involve a large-scale usability trial. The second was an empirical evaluation with a range of target users. To test whether the set of five BiDaML diagram types were sufficient to cover a wide range of real-world data analytic scenarios, we ran a group user study in which participants applied BiDaML to model a data analytics solution of their choosing based on their own experience. To test our hypothesis that BiDaML diagrams are suitable as a common form of

communication, we provided each participant with a BiDaML diagram created by another participant and asked them to compare it to a non-BiDaML diagram used as a control. To understand how easy BiDaML diagrams are to learn and use for specific projects, we also conducted a cognitive walkthrough of the tool with individual target end-users including data scientists and software engineers. Finally, we summarize our findings in using BiDaML in three industrial use cases.

4.1 Physics of notations evaluation

Physics of Notations analysis involves a detailed symbol-by-symbol analysis against design principles. There are two aspects to evaluate: the semantics of the language, and the visual notations used to represent these semantics. Moody states that "the principles [of PoN] define desirable properties of visual notations, which can be used for evaluation and comparison" and that they have "been successfully used to evaluate and improve three leading SE notations" [20]. In our case, we apply PoN to both improve and evaluate the cognitive effectiveness of the BiDaML visual notations. PoN-based visual language evaluation and/or improvement has been used in many studies, such as [21-24]. For purposes of space, only the results of the analysis will be provided here.

Semiotic clarity specifies that a diagram should not have symbol redundancy, overload, excess and deficit. All visual symbols in BiDaML have 1:1 correspondence to their referred concepts. A partial exception to this principle in BiDaML is tasks and subtasks, which are conceptually identical but use different symbol variants in the brainstorming diagram to ensure that the high-level tasks are visually salient. **Perceptual discriminability** is primarily determined by the visual distance between symbols. All symbols in BiDaML use different shapes as their main visual variable, plus redundant coding such as color and/or textual annotation. As color is only used redundantly, BiDaML remains suitable for handwritten diagrams and users with color blindness. **Semantic transparency** identifies the extent to which the meaning of a symbol can be inferred from its appearance. In BiDaML, icons are used to represent visual symbols and minimize the use of abstract geometrical shapes. **Complexity management** restricts a diagram to have as few visual elements as possible to reduce its diagrammatic complexity. We used hierarchical views for representation in BiDaML. For example, the brainstorming diagram supports recursive decomposition of tasks into subtasks, with low-level details of subtasks presented using technique diagrams. As future work, we will add a feature for users to be able to hide visual details for complex diagrams. **Cognitive integration** identifies that the notations should support the user to assemble information from separate diagrams into a coherent mental representation of a system; and it should be as simple as possible to navigate between diagrams. All the diagrams in BiDaML have a hierarchical tree-based structure relationship as explained in Section 3. Furthermore, the five BiDaML diagram types share symbols with each other (e.g. the same symbol for tasks) in order to facilitate cognitive integration of the different diagrams.

Visual expressiveness defines a range of visual variables (position, shape, size, brightness, color, orientation texture) to be used, resulting in a perceptually enriched representation that exploits multiple visual communication channels and maximizes computational offloading. BiDaML diagrams utilize position, shape, size, brightness and color to distinguish symbols and convey meaning. Orientation was not utilized due to the rotational symmetry of shapes used by BiDaML. Texture was only partially utilized (e.g. different line styles) due to limited tool support for texture in MetaEdit+ Workbench and existing diagram editing tools. **Dual coding** means that textual encoding should also be used, as it is most effective when used in a supporting role. In BiDaML, all visual symbols have a textual annotation. **Graphic economy** discusses that the number of different visual symbols should be cognitively manageable. As few visual symbols as possible are used in BiDaML; the most complex type of BiDaML diagram is the process diagram which defines 10 symbols in contrast to the 171 possible symbols [25] in BPMN 2.0 process diagrams. **Cognitive fit** means that the diagram needs to have different visual dialects for different tasks or users. However, as BiDaML borrows concepts and symbols from common notations where possible, such as mindmaps (brainstorming), BPMN (process diagram), and dataflow diagrams (data diagram), the BiDaML notation is expected to be immediately usable and familiar to a broad range of users without the need to introduce dialects. Nevertheless, in the future, we plan to provide different views for different users in our BiDaML support tool, and users will be able to navigate between views based on their requirements.

Table 1. BiDaML brainstorming diagram notations evolution: before and after several rounds of conducting physics of notations.

Diagram	Initial Notations	Final Notations
Brainstorming		

The initial and final notations designed for the brainstorming diagram after several rounds of conducting physics of notations are shown in table 1. The initial brainstorming notation was based on using two connectors, one to connect tasks and one to connect tasks to information. However, since it was not necessary to differentiate between different connectors, we consider the same connector type for both data and information in the updated version, to reduce the number of symbols, as recommended by physics of notations guidelines. Moreover, in the initial notation, the same shapes with different colors were used to differentiate high-level tasks from low-level tasks. This was potentially a problem for people with certain forms of color vision deficiency (color-blindness), and therefore, in order to avoid using many symbols and at the same time differentiate between high-level and low-level tasks, we automatically change the task symbol to a darker color with a pin icon if the task is directly connected to the problem symbol. High-level tasks are designed to resemble pinned todo-notes in order to improve semantic transparency. Tasks and subtasks both use an ellipse shape to show that they are conceptually similar.

The initial and final notations designed for the process diagram after several rounds of conducting physics of notations are shown in table 2. Our initial process diagram included 21 symbols and connectors, that contradicted PoN recommendation of around six symbols for novice users. We have now removed the unnecessary symbols and cut down the number of symbols and connectors to 10. The unnecessary items that are removed in the final version include different layers, operations, and icons to reflect datasets, models, domain knowledge, insights, and applications. Different layers correspond to the operations (BusinessOps, DataOps, AIOps, and DevOps) and since they are already specified in the brainstorming diagram, they were not adding any values to the process diagram. Moreover, the data items and artifacts are all specified in the data diagrams, and therefore using different icons to show how and where they are transmitted was only leading to complications in the process diagram. This greatly helped to reduce the number of notations and simplifying the process diagram. Given that the task icon is reused from the brainstorming diagram, and that the start, end, and condition icons are completely problem independent and shared between many diagrams, it is not a problem for novice users to learn these symbols. Moreover, the process diagram reuses symbols and syntax from BPMN where possible, so users who already have BPMN symbols and syntax stored in their long-term memory will only need to memorize the BiDaML specific aspects. The operation (task) symbol is also changed to an ellipse so that it matches and re-uses the task symbol in the brainstorming diagram. Furthermore, in the initial notation, the only difference between In-Organization and Out-Organization connectors was whether or not they span different organizations. However, since this can already be read off the process diagram by checking whether the connector spans different organization pools, in the final version, the notation is simplified by removing different types of organization connectors to avoid contradicting semiotic clarity.

Table 2. BiDaML process diagram notations evolution: before and after several rounds of conducting physics of notations.


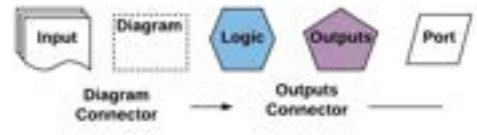

Diagram	Initial Notations	Final Notations
Process		

The initial and final notations designed for technique diagram after several rounds of conducting physics of notations are shown in table 3. The technique diagram initially consisted of only four symbols that left some room for us to extend it further with additional concepts/symbols. e.g. which techniques worked, issues faced (overfitting, etc.), workarounds (possibly leading to further issues), etc. Therefore, we added two new symbols to reflect whether the technique was successful or needs further work.

Table 3. BiDaML technique diagram notations evolution: before and after several rounds of conducting physics of notations.

Diagram	Initial Notations	Final Notations
Technique		

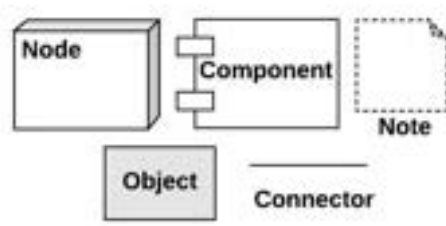

Table 4. BiDaML data diagram notations evolution: before and after several rounds of conducting physics of notations.

Diagram	Initial Notations	Final Notations
Data	<p style="text-align: center;">Data</p>  <p style="text-align: center;">Output</p> 	

The initial and final notations designed for the data diagram after several rounds of conducting physics of notations are shown in table 4. BiDaML initially had two sets of diagrams, one to reflect the set of data items and artifacts used or generated throughout the process, and the other for showing the outputs and reports to be generated as results of the technique and data diagrams. We have now merged these two as one single data diagram and added a new notation for “report”, to use for designing the expected reports and outputs. Also, the initial symbols were mostly rectangular and in the final version, we used a greater range of colors and shapes to distinguish between symbols. Moreover, data and task connectors are now considered as one connector. Furthermore, notations are defined in a more semantically transparent way, e.g. clipboards to represent reports.

The initial and final notations designed for the deployment diagram after several rounds of conducting physics of notations are shown in table 5. The initial deployment diagram, borrowed from the UML deployment diagram, used shades of grey to represent different symbols; however, the PoN guidelines advocate for utilizing color (so long as it is not the only means to distinguish symbols). In the updated version, we used a system layer diagram to show layers of the technology stack on the vertical y-axis rather than using a UML deployment diagram (as big data requires thinking in terms of distributed cloud platforms, services, and frameworks rather than individual nodes/devices). The new deployment diagram reuses symbols from other BiDaML diagrams to improve perceptual integration with the other diagrams. BiDaML data diagram notations are reused to help more precisely distinguishing the type of object without the need to learn new symbols. The updated version also shows how the services and infrastructure support (DevOps) tasks over the lifespan of the project, to answer questions such as “what infrastructure will the Data Science team use to (re-)train their models?” and “where will this model be deployed?”

Table 5. BiDaML deployment diagram notations evolution: before and after several rounds of conducting physics of notations.

Diagram	Initial Notations	Final Notations
Deployment		

Finally, the initial and final notations designed for showing the overview of all the diagrams are shown in table 6. Moody describes a “summary (long shot) diagram” as an “important mechanism to support conceptual integration” of diagrams. BiDaML’s initial overview diagram was at a meta-level (i.e. how diagrams connect and their cardinalities) rather than conveying the important problem specific information. Therefore, each diagram symbol was only capable of linking to a single diagram and was not sufficient to depict multiple diagrams of the same type. In the updated version, individual diagram instances are shown in the overview diagram to provide a “helicopter view” of the solution. Different colors and icons are used to represent the type of specific diagrams.

Table 6. BiDaML overview notations evolution: before and after several rounds of conducting physics of notations.

Diagram	Initial Notations	Final Notations
Overview		

4.2 Cognitive walkthrough

We conducted two sets of user studies, the first aiming at evaluating the BiDaML notations and comparing them with existing practices, with the second one evaluating the tool for specific projects.

4.2.1 Group user study

In this study, we asked a group of 12 data analysts, data scientists and software engineers to use any modeling language including UML, BPMN, data flow, entity relationship, mindmap, textual description, or their own ad hoc notation to model and describe a project of their choice on a piece of paper. We then introduced the BiDaML concept, notations, and diagrams and asked them to use one of the five BiDaML diagrams most related to the diagram they chose in the initial step to model the same problem on another piece of blank sheet. We then collected the handwritten diagrams and distributed them randomly between participants while making sure no one received their own handwritten diagram¹. In this step, we asked participants to fill in a questionnaire and decide which diagram they prefer for a business owner, business analyst, a data scientist, a software engineer, or themselves. They were also asked to rate whether BiDaML is easy to understand in this step. Finally, a demo of the BiDaML tool was given to the audience and they were asked to rate whether they felt the tool was easy to learn.

The group study consisted of 9 PhD students, 2 academic staff, and 1 participant from industry. 8 participants categorized themselves as software engineers, 5 as data analysts/scientists, and 1 as “other” (2 of the participants described themselves as both software engineers and data analysts/scientists). The distribution of data analytics/data science experience was: 6 participants with less than 1 year; 3 participants with 2 years; 1 participant with 4 years; and 2 participants with 5 to 9 years. The distribution of programming experience was: 1 participant with 2 years, 1 participant with 3 years, 2 participants with 4 years, 6 participants with 5 to 9 years, and 2 participants with 10 or more years.

Surprisingly, none of the participants’ initial diagrams used UML or BPMN. The majority of participants either made up their own ad-hoc notation, or combined multiple notations such as mindmaps, data flow diagrams, entity relationship, and textual descriptions.

Figure 20 shows participants’ responses for which of the two diagrams (handwritten initial diagram or other participant’s handwritten BiDaML diagram) they preferred. For the purposes of comparison, we excluded cases where participants responded “either”, “neither” or left the question blank.

Of the 9 participants that stated a preference, all responded that they would prefer BiDaML over the alternative to communicate a data analytics solution to a business owner. Despite the small number of responses, we can statistically generalize that BiDaML is preferable (>50%) to alternatives for communicating data analytics solutions to business owners, as confirmed by a two-sided binomial test ($p < 0.005$). A preference for BiDaML over alternatives was also weakly significant for communicating with software engineers ($p < 0.05$). While the small number of responses prevents us from drawing conclusions about the use of BiDaML for the other audiences, in all cases at least 6 of the 12 participants preferred the BiDaML diagram over the alternative. A Wilcoxon signed-rank test confirms that participants prefer BiDaML for a greater number of audiences than the alternative ($p < 0.005$), thus suggesting that BiDaML is suited for communication between different types of users in a data analytics project.

¹ As one participant returned from a break after diagrams were redistributed, they were allocated a pair of handwritten diagrams created prior to the study as a special case.

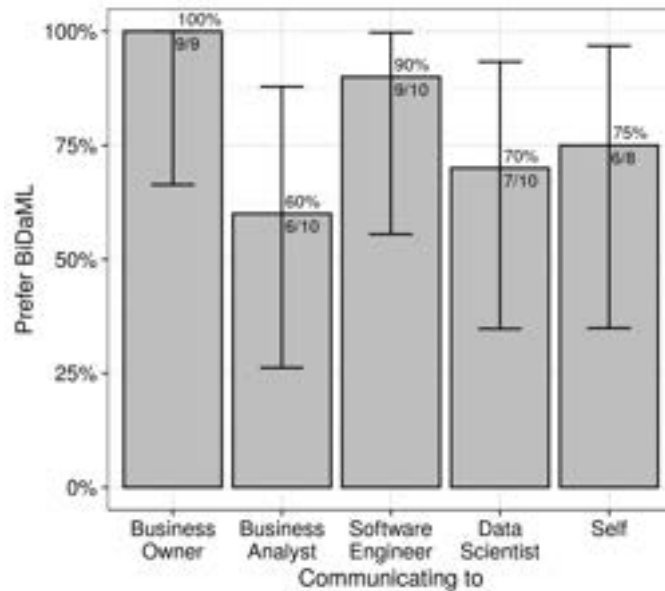


Figure 20. Percentage of participants that preferred handwritten BiDaML diagram over the handwritten alternative “to communicate a data analytics solution” for each audience. Error bars denote 95% confidence interval.

Figure 21 shows that 11 of 12 participants agreed that BiDaML was straightforward to understand. The one disagreement was likely unintentional, as the participant contradicted their response in a comment, stating that the diagrams “are easily understandable and communicable” and that BiDaML “includes a wide range of diagrams to cover almost all aspects of data and software engineering”.

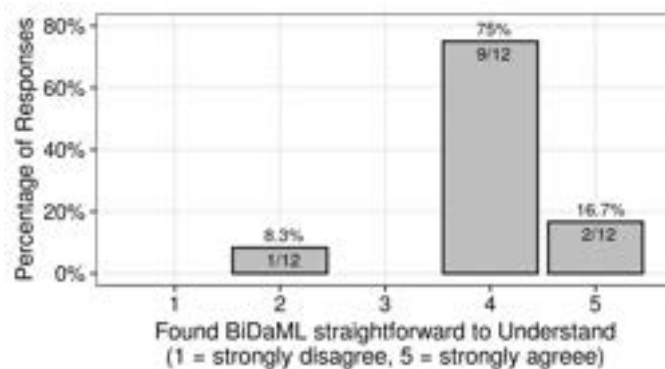


Figure 21. Distribution of participant agreement with the prompt “I found the BiDaML notation and diagrams straightforward to understand”.

Figure 22 shows that 10 of 11 participants agreed that the BiDaML tool was straightforward to learn. The reason for lack of stronger agreement appears to be due to the unpolished nature of the current BiDaML user interface implemented within MetaEdit+, with explanations including “a bit hard to use the MetaEdit+ [user interface], but it is easy to know how to use” and that it was “unclear in what way automated codes are generated”.

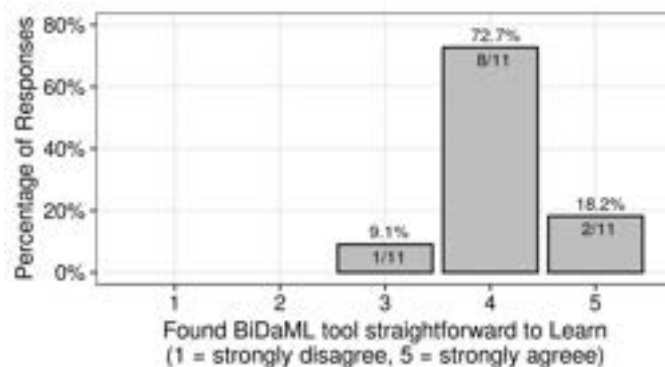


Figure 22. Distribution of participant agreement with the prompt “I found the BiDaML tool straightforward to learn”.

4.2.2 Individual user studies

The individual study [10] consisted of 1 research fellow in data science, 3 academic staff, 1 in data science and 2 in software engineering, and 1 data scientist from industry. We asked the three data scientists and two software engineers (all experienced in big data analytics tasks) to carry out a task-based end-user evaluation of BiDaML. The objective was to assess how easy it is to learn to use BiDaML and how efficiently it can solve the diagram complexity problem. BiDaML diagrams were briefly introduced to the participants who were then asked to perform three predefined modeling tasks. The first was to design BusinessOps, DataOps, AIOps, or DevOps parts of a brainstorming diagram for a data analytics problem of their choice from scratch. In the second, the subject was given a process diagram and asked to explain it, comment on the information represented and provide suggestions to improve it. The third involved subjects designing a technique diagram related to a specific task of the data analytics problem they chose for the first part of the evaluation. Figure 23 shows (a) devOps part of one of the brainstorming diagrams and (b and c) two of the technique diagrams that the data scientists/software engineers drew to help explain their current work tasks of (a) automated test-case generation (b) data wrangling and (c) data augmentation.

Overall, user feedback indicated that BiDaML is very straightforward to use and understand. Users felt they could easily communicate with other team members and managers and present their ideas, techniques, expected outcomes and progress in a common language during the project and before the final solution. They liked how different layers and operations are differentiated. Moreover, they could capture and understand business requirements and expectations and make agreements on requirements, outcomes and results through the project. These could then be linked clearly to lower-level data, technique and output diagrams. Using this feedback we have made some minor changes to our diagrams such as the shape and order of some notations, and the relationships between different objects.

However, several limitations and potential improvements were also identified in our evaluations. Some users would prefer to see technique and data diagrams components together in a single diagram, while others would prefer to have these separate. In the process diagram, some users would prefer to only see the operations related to their tasks and directly related tasks. Finally, one of the users wanted to differentiate between tasks/operations that are done by humans versus by a tool. In future tool updates, we will provide different views for different users and will allow users to hide/unhide different components of the diagrams based on their preference. Moreover, in our future code generation plan, we will separate different tasks based on whether they are conducted by human or tool. We will then run a larger user evaluation with business and domain expert end-users.



Figure 23. Example brainstorming and technique diagrams from our evaluators.

4.3 Industry Experiences with BiDaML

In this Section, we describe our experiences working with three different industry partners to model and capture the requirements of their big data analytics applications using BiDaML. A full list of the diagrams and the generated reports for the three projects are available in [18]. Based on the feedback from our industry users, as stated below, BiDaML can help understanding and communicating the project and its requirements within the teams, that can ultimately lead to reducing the time of the project’s development and increasing the reusability of the projects through a common platform. However, since the code generation part was the secondary contribution of this paper, a comprehensive evaluation of the code generation part remains the focus of our future work.

4.3.1 ANZ REALas (realas.com)

REALas is a property price prediction website owned by the Australia and New Zealand Banking Group (ANZ). Launched in 2011, REALas claims to provide Australia’s most accurate price predictions on properties listed for

sale². In this use-case, a complex new model needed to be developed to improve accuracy and coverage of the property price prediction model. The project team originally comprised a project leader, a business manager, a product owner, three software engineers, and one data scientist. There was an existing working website and ML model, as well as a dataset purchased from a third party. Two new data analyst/scientists were appointed to this project in order to create new models and integrate them with the existing website. The solution had initially been developed without the use of our BiDaML tool, and the challenges the team faced to communicate and collaborate through the process, was a key motivation for our research. The new data scientists initially lacked an understanding of the existing dataset and solution as well as domain knowledge. Therefore, it took them some time to be able to start the project. Moreover, communicating progress to the business manager and other members of the team was another problem. Together with the REALas team, we have used BiDaML to document the process from business analysis and domain knowledge collection through to the deployment of the final models in software applications.

We used BiDaML to redesign the whole project, in order to communicate and collaborate through the project, as well as automatically document all the above development steps. A brainstorming diagram was designed in the first place to help all the stakeholders fully understand and communicate the steps and existing solutions through a visualized drag-and-drop based platform. Once agreements were made on all the steps, a process diagram was created to specify the tasks to the existing stakeholders. These two steps only took a few hours from the whole team, and they could also incrementally modify these two during the process as needed. Data scientists were now fully aware of the domain knowledge, background of the project and existing models and could further leave comments and ask questions if further information was required. In the next step, data scientists worked on the data and ML parts, however this time, needed to visually record and keep track of the data items, artefacts, models, reports, etc that they tried, whether they were successful, failed, or were just being planned. This made it easy for them to communicate their progress and also reach agreement on the results. Our BiDaML tool code generation feature helped data scientists start from a template to work on instead of starting from scratch. The tool gives data scientists the ability to embed their code templates for future usage, as well as for documentation.

They gradually developed new models, and finally, worked with software engineers on a deployment diagram to define where and how to deploy the items generated in different steps. The new method was efficient in the way it took less time for the stakeholders to communicate and collaborate, and the step-by-step automatic documentation made the solution reusable for future reference. Based on the product owner's feedback *"this tool would have been helpful to understand and communicate the complexity of a new ML project within an organisation. It would assist the wider team to collaborate with data scientists and improve the outputs of the process"*.

4.3.2 Monash/VicRoads

VicRoads, the Victorian road traffic authority, utilizes the Sydney Coordinated Adaptive Traffic System (SCATS) to monitor, control and optimize traffic intersections. The Civil Engineering department at Monash University sought to build a traffic data platform that would ingest a real-time feed of SCATS data and integrate it with other transport datasets such as public transport travel history and traffic incidents reported through social media. Initially, the Civil Engineering department consulted with a software outsourcing company, who proposed a platform composed of industry standard big data tools. However, the software outsourcing company lacked understanding of the datasets and intended use of the platform, thus were unable to begin work on the project. Furthermore, it was unclear who would maintain the computing infrastructure, monitor data quality, and integrate new data sources after the initial phase of the project. We have worked with transport researchers and used BiDaML to document the intended software solution workflow from data ingestion through to traffic simulation and visualization. This allowed us to assist in formation of an alternative software solution that made better use of the systems and services already available.

We performed in-depth interviews with the project leader and traffic modeling expert, then used BiDaML to document the entire data analytics workflow including data ingestion, transport modeling, and result visualization. The traffic prediction brainstorming diagram was initially created as a handwritten sketch on paper, then later recreated using the BiDaML tool. The process diagram, technique diagram, data diagram, and deployment diagram were created directly using the BiDaML tool. While most BiDaML diagram types took only 15-30 minutes to create, the BiDaML process diagram proved the most time consuming, taking almost 3 hours due to the need to detail tasks to integrate each system and determine roles of individuals (we have since simplified the BiDaML process diagram notation in order to streamline the process). As BiDaML forces the user to consider all phases of the project, the modeling process helped reveal gaps in planning that required attention. Notably, no budget or personnel had been assigned to maintain the system after initial deployment, integrate new data sources, and monitor data quality/security. Indeed, in the BiDaML process diagram, we were forced to label both the organization and participant for these tasks as To Be Determined (TBD).

We presented the diagrams to the traffic modeling expert for feedback. This took place over a course of an hour session, in which we presented each diagram in the BiDaML tool. The BiDaML tool supported live corrections to the diagrams such as creation, modification or re-assignment of tasks as we discussed the diagrams with the traffic

² <https://realas.com/realas-science>

modeling expert. Feedback from the expert was positive: *“I think you have a good understanding of the business... how do you know about all of this? I think this is very interesting, very impressive what you are proposing. It covers a lot of work that needs to be done.”* While the expert stated that the BiDaML diagrams were helpful to *“figure out all the processes and what tasks need to be done”* they were reluctant to use BiDaML to communicate with external stakeholders in other organizations: *“to use this tool, it will be likely not possible, because they [the other organizations involved] have their own process, they don't want to follow a new one.”* We subsequently presented printouts of the diagrams to the project leader. The project leader expressed some uncertainty about the purpose of the notation; however, noted that an adaptation of the BiDaML data diagrams as a means to document data provenance (i.e. the ability to trace the origins of data analysis results back to the raw data used) would be *“very useful”*.

4.3.3 The Alfred Hospital (alfredhealth.org.au)

In this project, a group of radiologists, researchers and executives from the Alfred Hospital planned to use AI for predicting Intracranial hemorrhage (ICH) through CT Scans, work traditionally done by radiologists. The AI platform would enable them to prioritize the CT Scans based on the results and forward them to the radiologist for an urgent double check and follow up. Hence, a CT Scan with positive outcome could be reported in a few minutes instead of a few days. The team wanted to analyze the data before and after using the AI platform and based on the turnaround time (TAT) and cost analysis decide whether to continue using the AI platform or not. However, due to the diversity of the team it was difficult to communicate the medical terms to the data analysts and software engineers, and the analysis methods and software requirements and solution choices to the radiologists and the executive team. Thus, we used BiDaML with clinical, data science and software team members to model and document the steps and further plan for the next stages of the project.

We briefly introduced BiDaML to the team and since it seemed to be a well-designed fit for the project due to the diverse nature of the stakeholders, we decided to use the tool to model and analyze the requirements and capture the details. We had an initial one-hour meeting with one of the radiologists and started developing the models and collecting a deep understanding of the project, requirements, concepts, and objectives through the brainstorming diagram. Then we spent almost 30 minutes to document the entire data analytics workflow including data collection and wrangling, comparing the methods, making the final decision and deploying the final product through BiDaML's process, technique, data and deployment diagrams. Since we needed to deeply think about all the details and plans, BiDaML forced us to consider all phases and details of the project. We then organized a follow-up meeting with the team from the Alfred hospital, including two radiologists and the team leader and presented the diagrams for their feedback. The meeting took 30 minutes. During the meeting we modified the organizations and users involved as well as the expected reports and outcomes and the infrastructure in the deployment diagram. Going through the diagrams made us think about these and plan for them. We then shared the automatic report generated from the BiDaML tool with the team for their feedback. Feedback from the team was that *“BiDaML offered a simplified visual on different components of the project. These diagrams could be circulated to the project team and would clarify the workflow, requirements, aims and endpoints of each role and the entire project. In large-scale projects, BiDaML would be of even greater benefit, with involvement of multiple teams all working towards a common goal.”* However, *“The user interface seemed quite challenging to navigate. However, this could be easily negated with appropriate training and instructional material.”*

4.4 Threats to Validity

Due to the small number of participants involved in the group user study, only strong effects where nearly all participants agreed could be confirmed with any statistical significance. The group study questionnaire was delivered through an anonymous online form in order to encourage truthful feedback; however, it is possible that participants may have been biased to favor BiDaML diagrams due to familiarity with the researcher conducting the study. A further limitation is that time-bound user studies are unable to assess BiDaML at scale as a tool for communication within large projects. The authors have successfully utilized BiDaML to model mid-sized industrial projects such as property price prediction, transport data management, and CT Brain project communication. However, future work is needed to trial BiDaML in large long-term industrial projects. Moreover, the participants involved in the group user study were primarily PhD students and academics, so their attitudes to BiDaML may not reflect those of practitioners working in industry. This was the same for most of the participants (4 out of 5) in the individual study. However, it is not uncommon for industry to seek out PhD students and academic researchers to work on industry data science problems. Furthermore, the rise of powerful open source data analytics tools (e.g. TensorFlow) means that data scientists in academia will share much of the same software as data scientists in industry.

Another limitation of the group user study is that participants created their BiDaML diagram after their initial non-BiDaML diagram, so it is possible that the process of creating the initial diagram contributed to a better second diagram rather than the BiDaML notation itself. To avoid this, we could employ two separate evaluator groups to lessen this effect, one group to use only other modeling tools while the second group utilize BiDaML and to compare the achieved models. The challenge here would have been if we ran separate groups for each language, it would be difficult to control for the complexity of the diagrams during the comparison part of the study. By running a single session where participants design both a BiDaML diagram and a diagram using another modeling language,

it ensures that both diagrams have the same complexity. Ideally, we would counterbalance the design through switching the order that the participants create their diagrams. However, the study places an extensive time burden on participants, with many commenting that the study was too long. Due to the burden the study places on participants, we chose not to repeat the study (future studies could trial a reversed/randomized ordering). Furthermore, many participants have had extensive prior formal training in formal languages (such as UML, etc.), but not BiDaML, so even if the ordering was reversed, it would not be a fair comparison. Moreover, participants selected problems that they already had familiarity with, for example, one participant based their initial diagram on a figure they had previously crafted for a research paper. One might argue that the complexity of models differs according to the selected projects, and therefore, how to manage this variability in the interpretation of the models and getting the results? We indeed compared BiDaML/Non-BiDaML models created by each of the participants for the same project by randomly distributing the BiDaML/Non-BiDaML models to the other participants. We believe that including a variety of projects has helped to compare BiDaML/Non-BiDaML for the same projects in different levels of complexity. Moreover, since the participants were from different fields and domains, by providing the participants with the freedom to choose their own project, we could make sure that the participants would only focus on the modeling part. If we were to choose the project for the participants to work on, the results could be biased toward our chosen project depending on its complexity and participants' familiarity.

In contrast to the group study, the individual user studies were conducted in a less structured manner, and as feedback was collected directly, it was not anonymous. For the individual user studies, the researcher sat with participants for 30 minutes to 1 hour to introduce the tool and create the diagrams. This enabled valuable feedback on the BiDaML notations themselves and the overall tooling approach to be gathered. However, due to the researcher's intervention, the usability of the current MetaEdit+ interface for new users of the BiDaML tool remains uncertain.

5 DISCUSSION

Our experiences gained from discussions with the teams involved in data analytics projects, domain specific visual languages development, the PoN evaluation, and user studies indicate the lack of a prior modeling tool and collaboration framework in data analytics application development. The fact that none of our participants initially used UML or BPMN diagrams, as shown in Figure 1, as their way of communicating their data analytics projects with the other participants, as well as struggling to communicate their ideas indicates the lack of a common language or framework to collaborate through interdisciplinary teams. One of the initial questions we received from some of the participants was how to model, communicate and explain their problems to the other participants given they routinely work with a group of people in the same field, which is far from the interdisciplinary data analytics software development teams had in mind. Interestingly, based on our comparison in the group study, none of the participants, even though they were mostly software engineers and data scientists, used UML or BPMN as their initial diagrams, despite their familiarity with these modeling languages. The majority of participants finally made up their own ad-hoc notation or used a combination of multiple notations i.e., mindmaps, data flow diagrams, entity relationship, and textual descriptions. The decision of participants to use ad-hoc notations or a combination of notations, despite their formal training in software engineering and data science, further suggests the lack of existing standardized notations that support modeling of real-world data analytics problems. Also, our PoN evaluations and user studies, have provided evidence for the efficacy and usability of our BiDaML visual notations and languages as a standard way of communicating and modeling data analytics projects. Although we are expecting a wider gap in communications between software engineers, data scientists, and business managers, a recent online survey of business analysts and software architects by Linden et al. [26] suggested that UML, BPMN, SysML, and ArchiMate are the main notations used by practitioners for conceptual modeling. As our study primarily surveyed software engineers and data scientists, future work is needed to determine how business analysts respond to BiDaML. Conducting similar studies on business managers and analysts remains future work.

Limitations and deficiencies predominantly related to the maturity of the tool and its implementation. The current prototype tool has been developed using the bespoke MetaEdit+ domain-specific modeling development tool, which has limitations in fulfilling our desired end state. Some of the notable challenges we faced during evaluations and user studies were that although BiDaML can be accessed by all the stakeholders in different geographical locations, our intervention has been required so far, since for end users to use the tool, they would require a MetaEdit+ trial license. It also lacks a web-based user interface that would allow users to more easily access the tool without installing the software. Moreover, using MetaEdit+ limited us in generating documentation, reports and source code. It also limited us in easily giving access to users and evaluating the tool. We are currently looking to reimplement BiDaML as a stand-alone web-based tool and equip it with a recommender system to recommend suitable resources, models, techniques and solutions to the users based on the modeled problem and objectives. Another challenge we faced was that while BiDaML is ready for the initial problem definition and requirement analysis part, users continue to use existing tools or programming language to develop the ML and application development parts once they have completed the requirement analysis, modeling and planning part of

the project. We aim to further develop BiDaML integrations for well-known existing tools to encourage users to continue using BiDaML through the entire development of the final product.

6 RELATED WORK

There are many data analytics tools now available, such as Azure ML Studio [5], Amazon AWS ML [6], Google Cloud ML [7], and BigML [8] as reviewed in [9, 27]. However, these tools only cover a restricted set of phases of DataOps, AIOps, and DevOps and none cover business problem description, requirements analysis and design. Moreover, most end-users in multidisciplinary teams have limited technical knowledge of data science and programming and thus usually struggle to use these tools.

Some DSLs have been developed for supporting enterprise service modeling and generation using end-user friendly metaphors. An integrated visual notation (EML) for business process modeling is presented and developed in [28] using a novel tree-based overlay structure that effectively mitigates complexity problems. MaramaAIC [29] provides end-to-end support between requirements engineers and their clients for the validation and improvement of the inconsistencies in requirements. SDLTool [30] provides statistician end-users with a visual language environment for complex statistical survey design and implementation. These tools provide environments supporting end-users in different domains. However, they do not support data analytics processes, techniques, data, and requirements specifications, and do not target end-users for such applications.

Scientific workflows are widely recognized as useful models to describe, manage, and share complex scientific analyses and tools have been designed and developed for designing, reusing, and sharing such workflows. Kepler [31] and Taverna [32] are Java-based open-source software systems for designing, executing, reusing, evolving, archiving, and sharing scientific workflows to help scientists, analysts, and computer programmers. VisTrails [33] is a Python/Qt-based open-source scientific workflow and provenance management system supporting simulation, data exploration, and visualization. It can be combined with existing systems and libraries as well as user developed packages/modules. Finally, Workspace [34], built on the Qt toolkit, is a powerful, cross-platform scientific workflow framework enabling collaboration and software reuse and streamlining delivery of software for commercial and research purposes. Users can easily create, collaborate and reproduce scientific workflows, develop custom user interfaces for different customers, write their own specialized plug-ins, and scale their computation using Workspace's remote/parallel task scheduling engine.

Different projects can be built on top of these drag-and-drop based graphical tools and these tools are used in a variety of applications and domains. However, they are specifically designed for data analytics applications, and also make it hard for end-users to use data analytics and ML capabilities and libraries. Toreador [35] is a project aiming to overcome hurdles preventing companies from reaping the full benefits of big data analytics by delivering an architectural framework and components for model-driven set-up and management of big data analytics processes. Toreador is a "big data as a service" tool that helps to set up the pipeline and later execute it with other tools. However, what makes Toreador different from BiDaML is that Toreador users are software engineers lacking big data expertise who use Toreador as a replacement for NoSQL databases such as Cassandra or HBase, data preparation utilities such as Paxata, and distributed, parallel computing systems such as Apache Hadoop, Spark or Flink. Toreador needs a data set to be uploaded and a list of the processes to be chosen to be able to generate a workflow; that is where BiDaML can help end-users to make decisions and agreements, allowing them to be clear on the datasets to use or the methods to choose.

We aimed to provide a suite of visual languages to cover all aspects of data analytics projects with integration between different diagram concepts, instead of the user using different diagrams with different notations for various aspects. Having a suite of visual notations would help reusing and sharing the notations between diagrams as well as the participants. Most existing modeling notations such as UML, BPMN, etc provide a general purpose modeling framework and none of the modeling tools provide a high level to low level specific and goal driven means for the users to be able to easily memorize and model all the steps including brainstorming, technique, deployment, etc. However, any of the other modeling languages, i.e., iStar 2.0 [36] can be adapted and merged with BiDaML in case they cover an aspect of the data analytics application development that has not been covered in BiDaML. The main novelty of this paper is to present modeling languages to cover, model, and document all different aspects of data analytics applications in order to generate, recommend and reuse solutions. We aim to incorporate and synthesize the best existing practices instead of proposing completely new visual languages that do not follow any principles. Therefore, this does not preclude using iStar or integrating iStar or other goal-directed requirements modeling notations into BiDaML in the future.

Finally, some software tools implement algorithms specific to a given graphical model such as Infer.NET [37]. This approach for implementing data analytics techniques is called a model-based approach to ML [38]. An initial conceptualization of a domain-specific modeling language supporting code generation from visual representations of probabilistic models for big data analytics is presented in [39] by extending the analysis of Infer.NET. However, it is in very early stages and does not cover many of the data analytics steps of real-world problems.

7 CONCLUSIONS

We have described a set of visual notations for specifying data analytics projects. Our set of DSLs, BiDaML, aims to provide a similar modeling framework for data analytics solution design as UML does for software design. It is comprised of five high- to low-level diagrammatic types. These diagrams represent both data- and technique-oriented components of a data analytics solution design. A Physics of Notations analysis and a cognitive walkthrough with several end-users were undertaken to evaluate the usability of BiDaML. We have also used our diagrams to model several complex big data analytics problems. Key findings include 1) nearly all participants agreed that the BiDaML notations/tool was straightforward to understand and learn, and 2) participants prefer BiDaML for supporting complex data analytics solution modeling than other existing modeling languages.

Our intended future work includes providing multiple view/elision support for large diagrams in our BiDaML modeling tool. In addition, we see considerable scope for providing back end integration with other data analytics tools, such as Azure ML Studio. Our tool can then be used at an abstract level during requirements analysis and design, and then connected to different tools at a low level from say Google, Microsoft or Amazon. Therefore, our DSLs could be used to design, implement and control a data analytics solution. Our BiDaML tool will also support modeling and code generation, together with collaborative work support in the future. Since big data analysis has the same steps, the code generation feature of our tool will provide a set of templates for handling different classes of systems in data analytics projects. These will be leveraged to integrate our tool with other data analytics packages.

ACKNOWLEDGMENTS

This work was supported by ARC Discovery grant DP170101932 and ARC Laureate Fellowship FL190100035.

References:

- [1] I. Portugal, P. Alencar, and D. Cowan, "A Preliminary Survey on Domain-Specific Languages for Machine Learning in Big Data," presented at the IEEE International Conference on Software Science, Technology and Engineering (SWSTE), Beer-Sheva, Israel, 2016.
- [2] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, "A Survey of Open Source Tools for Machine Learning with Big Data in the Hadoop Ecosystem," *Journal of Big Data*, vol. 2, no. 24, 2015.
- [3] C. E. Sapp, "Preparing and Architecting for Machine Learning," Gartner Technical Professional Advice2017.
- [4] J. B. Rollins, "Foundational Methodology for Data Science," IBM Analytics2015.
- [5] *Microsoft Azure Machine Learning Studio*. Available: <https://studio.azureml.net/>
- [6] *Machine Learning at AWS - Amazon AWS*. Available: <https://aws.amazon.com/machine-learning/>
- [7] *Predictive Analytics - Cloud Machine Learning Engine | Google Cloud*. Available: <https://cloud.google.com/products/machine-learning/>
- [8] *BigML.com is Machine Learning Made Easy*. Available: <https://bigml.com/>
- [9] H. Khalajzadeh, M. Abdelrazek, J. Grundy, J. Hosking, and Q. He, "Survey and Analysis of Current End-user Data Analytics Tool Support," *IEEE Transactions on Big Data*, vol. 5, 2019.
- [10] H. Khalajzadeh, M. Abdelrazek, J. Grundy, J. Hosking, and Q. He, "BiDaML: A Suite of Visual Languages for Supporting End-user Data Analytics," in *IEEE Big Data Congress*, Milan, Italy, 2019, pp. 93-97.
- [11] W. v. d. Aalst and E. Damiani, "Processes Meet Big Data: Connecting Data Science with Process Science," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 810-819, 2015.
- [12] D. D. Cock, "Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project," *Journal of Statistics Education*, vol. 19, no. 3, 2011.
- [13] D. Sculley *et al.*, "Hidden Technical Debt in Machine Learning Systems," presented at the 28th International Conference on Neural Information Processing Systems (NIPS), Montreal, Canada, 2015.
- [14] OMG. (2011). *Business Process Model And Notation (BPMN)*. Available: <https://www.omg.org/spec/BPMN/2.0/>
- [15] S. Ambler, *The Object Primer: Agile Model-Driven Development With Uml 2.0 3rd Edition*. Cambridge University Press 2004.
- [16] S. Kelly and R. Pohjonen, "Worst Practices for Domain-Specific Modeling," *IEEE Software*, vol. 26, no. 4, pp. 22-29, 2009.
- [17] *MetaEdit+ Domain-Specific Modeling tools - MetaCase*. Available: <https://www.metacase.com/products.html>
- [18] *BiDaML Case Studies*. Available: <http://bidaml.visualmodel.org/>
- [19] D. Moody, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756-779, 2009.
- [20] D. Moody and J. van Hillebergersberg, "Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams," in *Software Language Engineering*, Berlin, Heidelberg, 2009, pp. 16-34: Springer Berlin Heidelberg.
- [21] M. Famelis and M. Chechik, "Managing design-time uncertainty," *Software & Systems Modeling*, vol. 18, no. 2, pp. 1249-1284, 2019/04/01 2019.
- [22] H. Henriques, H. Lourenço, V. Amaral, and M. Goulão, "Improving the Developer Experience with a Low-Code Process Modelling Language," presented at the Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Copenhagen, Denmark, 2018. Available: <https://doi.org/10.1145/3239372.3239387>
- [23] T. Miranda *et al.*, "Improving the Usability of a MAS DSML," in *Engineering Multi-Agent Systems*, Cham, 2019, pp. 55-75: Springer International Publishing.
- [24] E. Gonçalves, C. Almendra, M. Goulão, J. Araújo, and J. Castro, "Using empirical studies to mitigate symbol overload in iStar extensions," *Software and Systems Modeling*, 2019/12/12 2019.
- [25] N. Genon, P. Heymans, and D. Amyot, "Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation," in *Software Language Engineering*, Berlin, Heidelberg, 2011, pp. 377-396: Springer Berlin Heidelberg.
- [26] D. van der Linden, I. Hadar, and A. Zamansky, "What practitioners really want: requirements for visual notations in conceptual modeling," *Software & Systems Modeling*, vol. 18, no. 3, pp. 1813-1831, 2019.
- [27] H. Khalajzadeh, M. Abdelrazek, J. Grundy, J. Hosking, and Q. He, "A Survey of Current End-user Data Analytics Tool Support," in *IEEE International Congress on Big Data 2018*, San Francisco, USA, 2018, pp. 41-48.
- [28] L. Li, J. Grundy, and J. Hosking, "A Visual Language and Environment for Enterprise System Modelling and Automation," *Journal of Visual Languages & Computing*, vol. 25, no. 4, pp. 253-277, 2014.

- [29] M. Kamalrudin, J. Hosking, and J. Grundy, "MaramaAIC: Tool Support for Consistency Management and Validation of Requirements," *Automated Software Engineering*, vol. 24, no. 1, pp. 1-45, 2017.
- [30] C. H. Kim, J. Grundy, and J. Hosking, "A Suite of Visual Languages for Model-Driven Development of Statistical Surveys and Services," *Journal of Visual Languages and Computing*, vol. 26, no. C, pp. 99-125, 2015.
- [31] B. Ludäscher *et al.*, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039-1065, 2005.
- [32] K. Wolstencroft *et al.*, "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud," *Nucleic acids research*, vol. 41, no. 1, pp. 557-561, 2013.
- [33] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Visualization Meets Data Management," in *ACM SIGMOD international conference on Management of data*, 2006, pp. 745-747.
- [34] P. W. Cleary, D. Thomas, M. Bolger, L. Hetherington, C. Rucinski, and D. Watkins, "Using Workspace to Automate Workflow Processes for Modelling and Simulation in Engineering," presented at the 21st International Congress on Modelling and Simulation, 2015. Available: <https://research.csiro.au/workspace/>
- [35] E. Damiani, C. Ardagna, P. Ceravolo, and N. Scarabottolo, "Toward Model-Based Big Data-as-a-Service: The TOREADOR Approach," in *Advances in Databases and Information Systems*, Cham, 2017, pp. 3-9: Springer International Publishing.
- [36] F. Dalpiaz, X. Franch, and J. Horkoff. (2016). *iStar 2.0 Language Guide*. Available: https://sites.google.com/site/istarlanguage/home#h.p_ID_44
- [37] T. Minka, J. Winn, J. Guiver, and D. Knowles, "Infer .NET 2.4, 2010. Microsoft Research Cambridge," 2010.
- [38] C. M. Bishop, "Model-based Machine Learning," *Philosophical Transactions of the Royal Society A, Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1984, 2012.
- [39] D. Breuker, "Towards Model-Driven Engineering for Big Data Analytics – An Exploratory Analysis of Domain-Specific Languages for Machine Learning," presented at the 47th Hawaii International Conference on System Science, 2014.

6

Human-centric DSL Modelling and Collaboration

6.1 Experiences developing architectures for realising thin-client diagram editing tools

Grundy, J.C., Hosking, J.G., Cao, S., Zhao, D., Zhu, N., Tempero, E. and Stoeckle, H. Experiences developing architectures for realising thin-client diagram editing tools, *Software – Practice and Experience*, vol. 37, no.12, Wiley, October 2007, pp. 1245-1283

DOI: [10.1002/spe.803](https://doi.org/10.1002/spe.803)

Abstract: Diagram-centric applications such as software design tools, project planning tools and business process modelling tools are usually ‘thick-client’ applications running as stand-alone desktop applications. There are several advantages to providing such design tools as Web-based or even PDA- and mobile-phone-based applications. These include ease of access and upgrade, provision of collaborative work support and Web-based integration with other applications. However, building such thin-client diagram editing tools is very challenging. We have developed several thin-client diagram editing applications realized as a set of plug-in extensions to a meta-tool for visual design environment development. In this paper, we discuss key user interaction and software architecture issues, illustrate examples of interacting with our thin-client diagram editing tools, describe our design and implementation approaches, and present the results of several different evaluations of the resultant applications. Our experiences will be useful for those interested in developing their own thin-client diagram editing architectures and applications.

My contribution: Co-developed key ideas for this research, co-designed approach, co-supervised the two Masters students and the research assistant, wrote majority of paper, co-lead investigator for funding for this project from FRST

Experiences developing architectures for realising thin-client diagram editing tools

JOHN GRUNDY^{1,2}, JOHN HOSKING¹, SHUPING CAO¹, DENJIN ZHAO¹, NIANPING ZHU¹,
EWAN TEMPERO¹ AND HERMANN STOECKLE¹

Department of Computer Science¹ and Department of Electrical and Computer Engineering²,
University of Auckland, Private Bag 92019, Auckland, New Zealand

{john-g, john, nianping, ewan, herm}@cs.auckland.ac.nz, zhuangcao@hotmail.com,
dzhao@ist.psu.edu

ABSTRACT

Diagram-centric applications such as software design tools, project planning tools and business process modelling tools are usually “thick-client” applications running as stand-alone desktop applications. There are several advantages to providing such design tools as web-based or even PDA- and mobile phone-based applications. These include ease of access and upgrade, provision of collaborative work support and web-based integration with other applications. However, building such thin-client diagram editing tools is very challenging. We have developed several thin-client diagram editing applications realised as a set of plug-in extensions to a meta-tool for visual design environment development. In this paper, we discuss key user interaction and software architecture issues; illustrate examples of interacting with our thin-client diagram editing tools; describe our design and implementation approaches; and present results of several different evaluations of the resultant applications. Our experiences will be useful for those interested in developing their own thin-client diagram editing architectures and applications.

KEY WORDS: thin-client diagramming, software architecture, web and mobile user interfaces, CASE tools

INTRODUCTION

Diagrammatic applications include design tools for a variety of domains such as information structuring and browsing, concept sketching and discussion, project management, and software and user interface design. Examples of diagram types include general ones, such as trees for depicting hierarchies and graphs for network, relationship and dependency specification together with more specialised domain-specific notations, such as Gantt and Pert charts for project management, or UML diagrams for software design. Traditional diagramming tools for such applications are built using thick client, window-based interfaces that limit their use to desktop and laptop computers. This approach provides highly responsive diagram editing and viewing facilities, can leverage sophisticated thick-client interaction techniques and enables management of information on local workstations^{15, 23, 6}. Disadvantages of this approach include the need to install and update software on every user’s workstation, the complexity and learning curve associated with using many tool interfaces, the complex, heavyweight architectures needed to support collaborative editing, and lack of support in most tools for modifying diagramming notations and semantics^{26, 30, 20}.

To improve access to these tools, thin client diagramming approaches (Baudisch et al 2004; Gordon et al 2003) have been proposed and prototyped. These typically use a web browser for viewing and sometimes editing of diagrams. Users view diagrams as content of a “web page” that also includes controls such as buttons and links for modifying the diagram or moving to other diagrams (Graham et al, 1999; Maurer and Holz, 2002). Diagrams may be constructed with combinations of automatic and manual layout, resizing, highlighting and so on. Technologies to render the diagrams include GIF images, SVG (Scalable Vector Graphics) renderings, and 3D VRML (Virtual Reality Modelling Language) visualisations. Potential advantages of this approach are a consistent look and feel across all web-based diagramming tools, use of web page design techniques to aid interaction and learning, limited install and update problems, and use of conventional web architectures to support multi-user collaborations. Devices such as wireless PDAs and mobile phones have also become very widely available and used in recent years. Many offer a range of sophisticated content including styled text, rich graphical images and full-motion video streaming. However, very few applications for mobile devices provide dynamic diagrammatic display and almost none interactive diagram editing.

We have been developing Pounamu, a meta-tool that provides very flexible diagram and meta-model specification techniques (Zhu et al, 2004). Pounamu is used initially to specify a new diagramming tool, and then interprets the

specification to provide the new tool's functionality. Both the metatool and generated tools have a thick-client interface. We wanted to provide Pounamu users the ability to access diagrams via web browsers, PDAs and mobile phones. To this end the Pounamu/Thin extension provides a thin-client diagramming infrastructure allowing users of web browsers to view and manipulate Pounamu diagrams.

Web browser-based user interfaces are more restricted than thick client interfaces in the types of interaction they allow. To investigate the degree to which these restrictions can be overcome, we have developed support for different interaction styles in Pounamu/Thin. One uses a GIF format and is usable by all browsers, but has limited interactivity, and a second uses SVG, which requires a separate plug-in, but allows more interaction. We report on the effectiveness of these two styles of thin-client diagramming user interface. To explore the issues involved in making thin-client diagramming applications available on mobile devices like PDAs and mobile phones we have also extended Pounamu/Thin to provide such support.

We begin by motivating our research using examples of Pounamu diagramming tools. We then review related research in the areas of thick-client diagramming infrastructure, thin-client diagramming approaches, and small-screen diagramming applications. We then demonstrate by examples how users interact with Pounamu/Thin applications using web browsers and mobile devices. The architecture, design and implementation of Pounamu/Thin are then described in detail. We describe our experiences building diagramming tools with Pounamu/Thin, summarise the results of usability evaluations and discuss the strengths and limitations of our approach. We conclude with a summary of the key contributions of this research and development.

MOTIVATION

The traditional approach to providing diagramming tools is to use a thick-client program. Our thick-client meta-tool, Pounamu, provides a set of tools for designing shape and connector appearance, meta-model elements, views of meta-model elements using shapes and connectors, and event handlers for specifying semantic behaviour (Zhu et al, 2004). Figure 1 (a) shows an example diagramming tool generated by Pounamu, a Unified Modelling Language (UML) CASE tool, in use. A thick-client interface is provided for all Pounamu tools, which includes an element tree (1), pop-up and pull-down (2) menus, drawing canvas (3), shape property editor, status window (4), and directly manipulable shapes (5) and shape elements. Figure 1 (b) shows a project management tool. These tools use thick-client interaction technologies and Pounamu must be installed on machines and periodically upgraded. A set of complex collaborative work supporting plug-ins have been developed which use peer-to-peer technologies to support synchronous collaborative diagramming and asynchronous version control.

Key advantages are highly responsive interaction when directly manipulating diagrams and the ability for tool developers to make full use of the graphics facilities available on the host computer. However, as noted earlier, this approach suffers the common thick-client application problems of: the need to install and update applications on all host computers; complex and difficult to learn user interfaces; and complex infrastructure to support multi-user collaborative work, especially synchronous diagram editing. As web-based user interfaces have become pervasive many applications that traditionally used thick-client interfaces have been reengineered with thin-client versions. Key design features of these web-based applications include a focus on simple, easy-to-use and consistent user interface designs, seamless support for collaborative work via the client-server approach inherent to web applications, and server-side integration with legacy systems and their facilities.

Our high level aim with Pounamu/Thin was to experiment with thin-client diagramming support for Pounamu-based diagramming applications to try and leverage these advantages in our tools. Similarly, to support pervasive access and mobile collaborative design, we wanted to make Pounamu modelling tools accessible on a range of mobile devices, such as mobile phones and PDAs. While the case for web browser based solutions is relatively easy to make, it is a more open issue as to whether the limited interfaces such mobile devices provide afford sufficient interaction capability for diagrammatic applications. We were interested in providing a proof of concept technology base to explore this, particularly as a number of industry partners we have been working with have expressed strong interest in providing mobile access to their toolsets. These real-world mobile applications included project planning (eg Gantt chart manipulation), health systems information capture and visualisation, and complex interactive enterprise reporting visualisations. In each case the desire was to allow quick access and update of strategic and tactical information in visually accessible and manipulable forms while end users were "in the field".

Our approach to providing such mobile device interactive diagramming support was to provide an optional plug-in extension to Pounamu allowing users the choice of accessing generated diagramming tools via a web-based, thin-client infrastructure instead of a thick-client. Our lower level aim was to explore the technical and usability issues associated with building complex diagram-based user interfaces for both web browsers and mobile devices. The technical solution

we chose to achieve this was to develop a set of web server components that support multi-user access to tools generated by Pounamu, through thin clients via web browsers and mobile devices.

Figure 1 (c) shows a web client user interface implemented by Pounamu/Thin. This requires no host machine installation (other than a generic web browser), has only a single Pounamu upgrade point on the shared web server, and uses a conventional multi-user, client-server web architecture to support multiple, collaborating users. Figure 1 (d-e) shows similar Class and Gantt chart diagrams viewed on a mobile phone. These thin-client diagramming applications provide the same range of diagram display and editing capabilities as the thick-client examples in Figure 1 (a and b). However, alternative mechanisms to access these capabilities are provided and the mobile device uses multi-level interactive zooming to manage the size and complexity of diagrams.

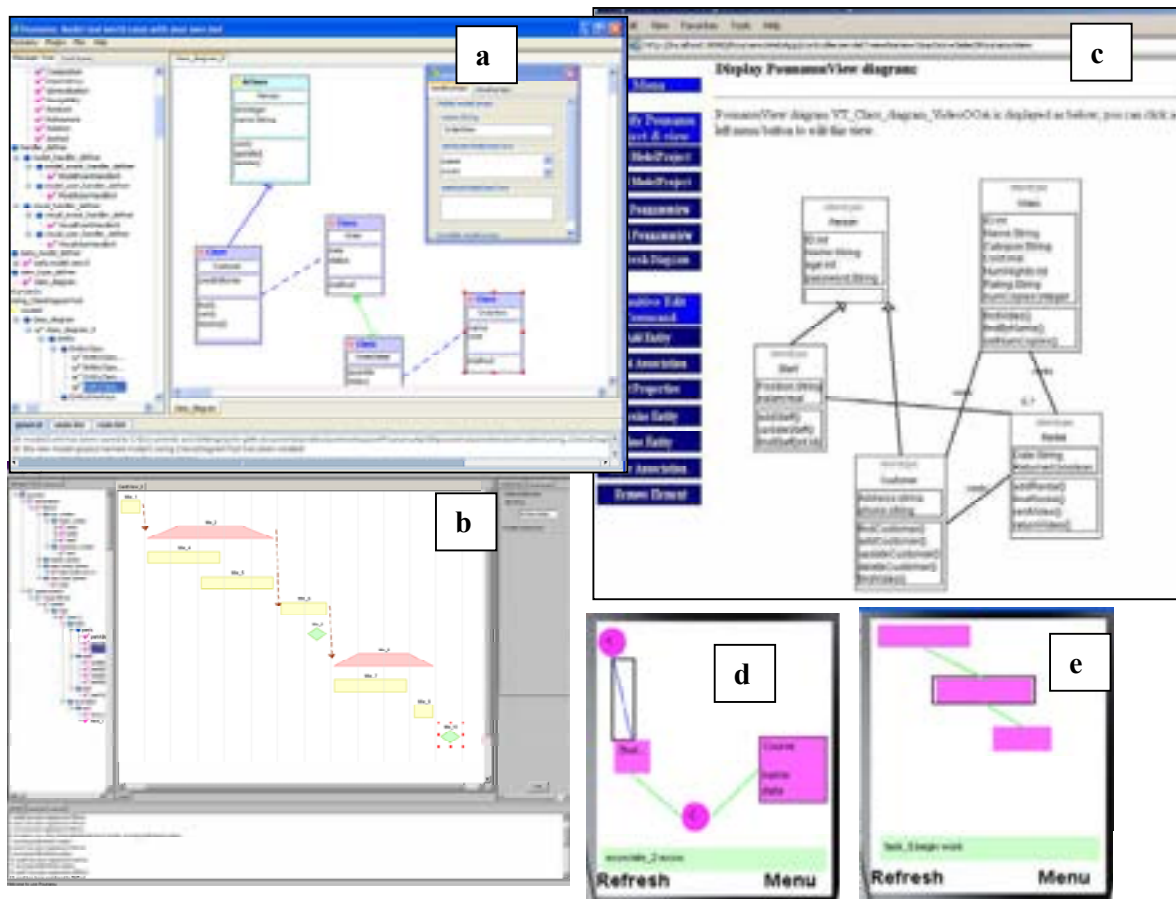


Figure 1. Examples of Pounamu diagramming tools.

Web browsers and in particular browsers running on mobile devices are constrained in a number of ways over desktop clients. This makes it particularly challenging to realize diagram-based interfaces on them. Some of these constraints include:

- Rendering of images such as those in Figure 1, often lacks the level of precision available in desktop application user interface libraries. This is particularly exacerbated with small screen devices.
- Manipulation of browser-displayed diagrams using only server-side technology means users can only click on the diagram and a location point sent back to the server for processing. This makes implementation of drag-and-drop style direct manipulation interfaces infeasible.
- Using client-side scripting to enable more interactive direct manipulation of diagram content exposes the application to a range of inconsistent “standards” for different browser scripting technologies or reliance on browser plug-ins.
- A range of web browser applications and versions, web browser plug-ins for diagram rendering, and a very wide range of mobile devices exist. This makes the issue of deploying applications onto devices running different

browser applications, versions of applications, plug-ins and operating systems complex, a situation we were attempting to avoid by adopting a thin client approach.

- While web browsers running on desktop machines have the potential to provide high performance most mobile devices have relatively low processor speed, small memory and storage high network latency and bandwidth cost.
- Navigation between diagrams displayed in a web browser on a desktop computer or a mobile device require either multiple browser windows or hyperlinks. In addition, for mobile devices zooming and elision techniques are required in order to display large diagrams on small screens and yet still keep the displayed diagram meaningful.
- Ideally a thin-client diagramming application should support user preferences so that different users can specify different diagram content rendering approaches, usage of client side vs server-side scripting, zooming and navigation configurations.
- Collaborative work by multiple users on the same diagrammatic content should be supported with appropriate access control, group awareness and versioning of diagrams. For ease of deployment and use the provision of such capabilities should use standard web client-server technologies rather than custom peer-to-peer or other approaches.

RELATED WORK

Applications that provide thick-client diagramming support have been extensively investigated. Examples of software design tools include Rational Rose™ (Quatrani and Booch, 2000), Argo/UML (Robbins et al, 1998), and JComposer (Grundy et al, 2000). More general diagramming packages and authoring tools include MS Visio™, PowerPoint™ and Photoshop™. More domain-specific tools include project management tools, such as MS Project™; XML Schema editors, such as XML Spy™; and custom visual language tools, such as LabView™, Prograph (Cox et al, 1989) and Forms/3 (Burnett et al, 2001). These all provide a thick-client user interface with each user having a copy of the tool on their desktop. The need to be able to tailor such design tools to different user's preferences and rapidly develop new design methods and tools has led to the development of meta-tools. Examples include MetaEDIT+ (Kelly et al, 1996), Kogge (Ebert et al, 1997), JViews (Grundy et al, 2000), MetaMOOSE (Ferguson et al, 2000), DiaGen (Minas and Viehstaedt, 1995), Vampire (McIntyre, 1995) and Escalante (McWhirter and Nutt, 1994). Despite the fact that each of these meta-tools provides a specification of the intended diagramming tool that could be interpreted to provide a thin-client version of the tool, none do so.

A wide variety of thin-client tools have been developed to exploit web-based delivery of information that have traditionally used non-diagram oriented thick-client applications. Some examples include: MILOS (Maurer and Holz, 2002; Maurer et al, 2000), a web-based process management tool; BSCW (Bently et al, 1995), a shared workspace system; software education environments (Chalk, 2000); Web-CASRE (Lyu and Schoenwaelder, 1998), which provides software reliability management support; and web-based tool integration approaches (Kaiser et al, 1998). A browser-based user interface for such applications allows users to access the tool via a URL, without the need for application download, install or update, at the expense of more sophisticated user login requirements. A consistent web interface look and feel across applications and browser-based integration of applications can also be facilitated. When diagrammatic content is supported, it is often limited to display-only, e.g. the display of UML diagrams derived from formal specifications in TCOZ (Sun et al, 2001). Most tools providing such thin-client diagram rendering have adopted custom algorithms and implementations to producing the diagrams with fixed diagram layout and appearance.

Most tools that provide web-based diagramming have used Java Applets or similar heavyweight browser plug-ins (Graham et al 1999). These have often suffered from reliance on particular plug-in versions and lack of consistency across multiple applications. Supporting collaborative work activities in such applications requires similar implementation effort as in thick-client tools. Some recent efforts at building true thin-client web-based diagramming tools include: Seek (Khaled et al, 2002), a UML sequence diagramming tool; NutCASE (Mackay et al, 2003), a UML class diagramming tool; and Cliki (Gordon et al, 2003), a thin-client meta-diagramming tool. These have given a proof-of-concept demonstration that web-based interfaces can be used to both visualise and edit specific kinds of diagrammatic content. All of these have used custom approaches to realise the thin-client diagramming tool. Limited tailoring of notations is supported in Cliki but not the other tools.

Thin-client diagramming systems have focused on supporting standard web browsers on desktop or laptop PCs, rather than smaller screen mobile PDA or phone devices. However some mobile applications provide diagrammatic representations of complex information, including for: tourism and travel planning, map usage, and management (Luz and Masoodian, 2004; Masoodian and Budd, 2004; Rossel, 1999; Stephanidis, 2001; Wobbrock et al, 2002). Most have read-only diagrams that do not support editing. Some provide pan and zoom-based displays enabling users to focus in on areas of interest quickly and navigate complex information spaces. However these capabilities are limited to either

the mobile device's standard characteristics or single points of zoom and focus. Several systems have been developed to assist in realising mobile, small-screen device user interfaces. Some provide overviews of complex information such as web pages using e.g zoomed-out copies of pages (Eisenstein and Puerta, 2001; MUPE 2006) or summarized versions of the page (Baudisch et al 2004; Chen et al, 2003). These "content outlining" approaches transform pages into sets of tiles (Buyukkoten et al 2000; Baudisch et al 2004) and while they assist navigation they are unsuitable for diagram representation. Some mobile browsers combine tiling with zooming, so that users can access individual tiles from an overview (Chen et al 2003; Eisenstein, 2000). These increase the amount of complex content that can be reasonably accessed and browsed on a small screen mobile phone. Approaches that collapse page content allow users to zoom into relevant areas by collapsing less relevant areas (Baudisch et al 2004) increasing their ability to access complex information that otherwise could not be presented on a single small screen.

A number of architectures have been developed to support "multi-device user interface" construction (Bonifati et al, 2000; Palm Corp., 2001; Grundy and Zhou, 2003; Van der Donckt et al, 2001; Marsic 2001). Techniques range from "clipping" services, i.e. translating content for conventional web browsers into other forms, to "generic" interface specifications, where a single specification may be realised on a wide range of devices, including standard web browsers and various small-screen devices. However most of these application interfaces have been limited to text and form-based data rather than richer diagrammatic content, typically translating from an abstract XML format into a device-specific format. Any diagramming support is usually implemented in a custom fashion specifically for each mobile device type. While this allows some optimisations to be made in terms of content production, it has the disadvantage of not being able to leverage work on diagramming meta-tools and architectures.

As thin-client applications are generally client-server systems supporting multiple concurrent users, they provide a natural collaborative work-supporting infrastructure. Many collaborative work-supporting thin-client applications have been developed, including BSCW (Bently et al 1995), MILOS (Maurer et al, 2002; Maurer et al, 2000) and various UML design tools (Graham et al, 1999; Mackay et al, 2003). Most support collaboration via awareness of others' activities and limited shared editing of content. As clients very frequently request content for display and editing in the client browser, the thin-client application architectures facilitate the implementation of collaborative work access control, synchronisation and group awareness capabilities. As they use standard internet protocols they are easy to deploy in organisations with firewalls and policies that prevent peer-to-peer approaches. Diagrammatic thin-client interfaces thus have the potential to support both awareness and content sharing by multiple users. However, there are potential disadvantages as well. These include a limited ability to push other users' changes to one's web browser from the server and limited drag-and-drop manipulation of content with synchronous notification of change to other users.

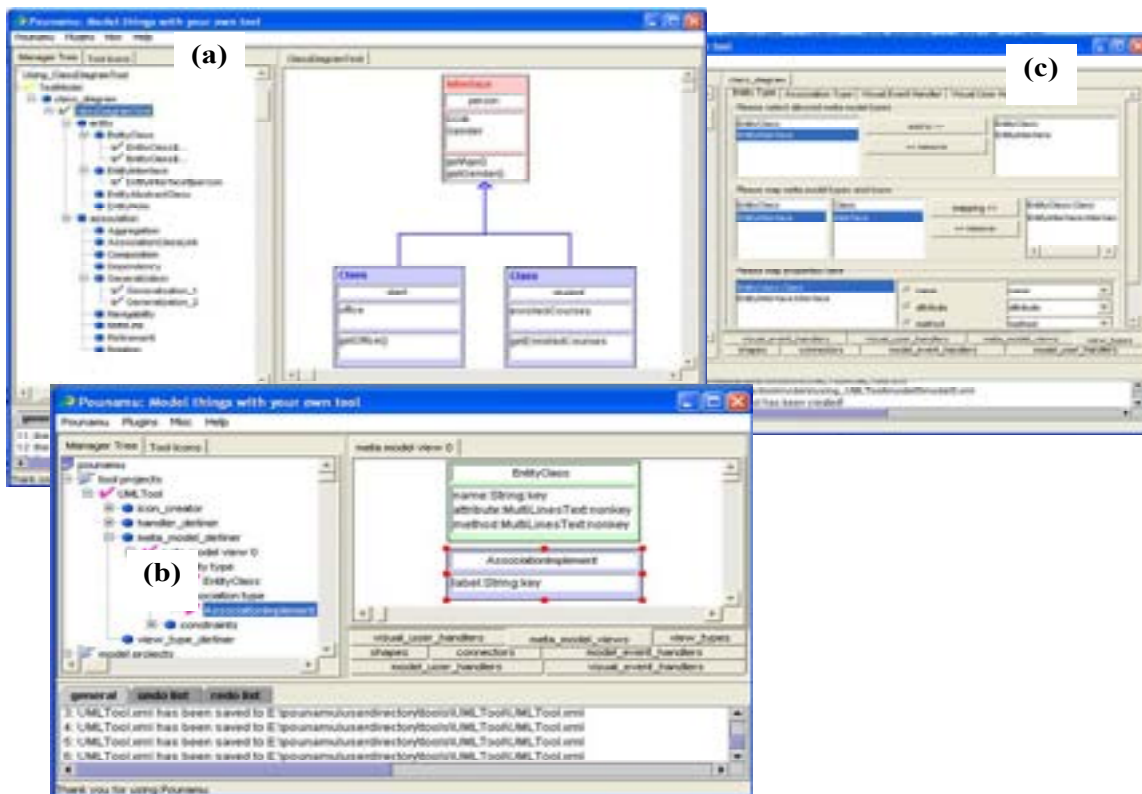


Figure 2. Examples of Pounamu meta-tool UML tool specifications.

OVERVIEW OF POUNAMU/THIN

In this section we illustrate the user interfaces of Pounamu/Thin via an exemplar application, a simple Unified Modelling Language (UML) design tool prototype. This tool, specified using Pounamu (Zhu et al, 2004), includes UML class, sequence, collaboration, and deployment diagrams. This is but one exemplar tool; as Pounamu is a metatool the tool's appearance and semantics can be changed on-the-fly by the user, or a completely new tool specified and used. We also emphasise that the Pounamu/Thin web components illustrated here are completely tool-independent i.e. no code changes are needed to support rendering and editing of any diagramming tool specified using our Pounamu meta-tool.

Figure 2 shows part of the specification of the UML tool using Pounamu. Figure 2 (a) shows the class icon shape being defined. Figure 2 (b) shows part of the UML tool meta-model specification (the entities, associations and various inter-relationships defined for the tool). Figure 2 (c) shows specification of the class diagram view (mapping icon shapes to entities etc). The Pounamu meta-tools are currently all realised via thick-client interfaces. Figure 1 (a), shows the generated UML tool also realised as a thick-client desktop application by Pounamu. To realise a tool the user simply selects its specification and Pounamu on the fly creates an appropriate editing environment for that specification. Users may change the tool specification while their editing tool is in use e.g. modify or add icon shapes; change meta-model types etc. These changes are reflected immediately in the running editing tools (Zhu et al, 2004).

Figure 3 shows the web client diagramming interface provided by Pounamu/Thin when viewing and editing an example UML class diagram with the above UML tool specification. This web browser-based user interface includes a set of buttons to control the project (add views, open views) (1); a set of buttons to manipulate the diagram (add shapes and connectors, move and resize elements, edit element properties) (2); a message area to provide feedback to users (3); and a diagram (4), which can be clicked on to manipulate it. A simple user authentication mechanism permits registered users to access and manipulate any Pounamu projects in a shared repository. This could readily be extended to project-specific or role-specific access.

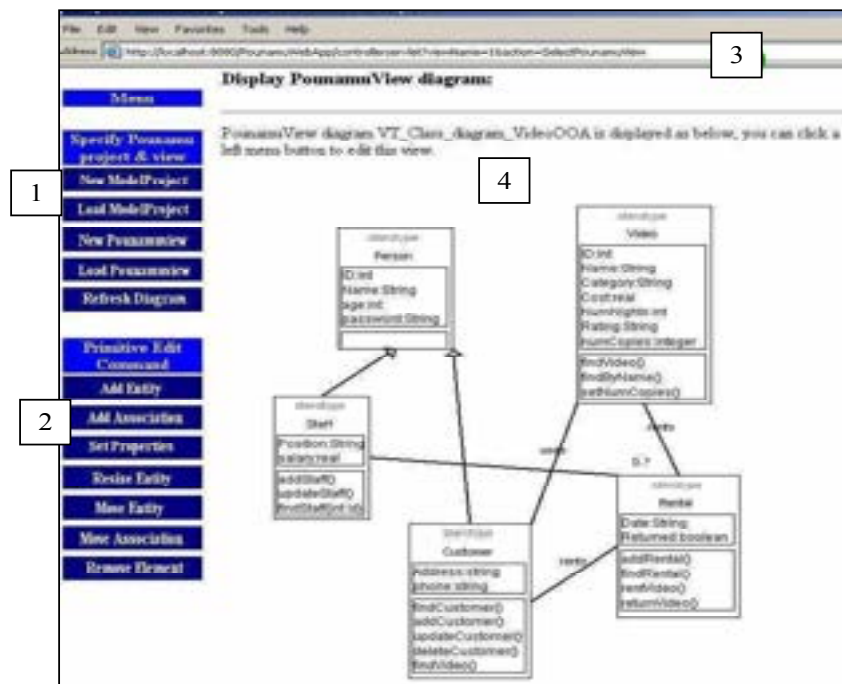


Figure 3. Example of using Pounamu/Thin.

With such a browser-based interface, there are a variety of approaches we could have taken to allow users to select and manipulate diagram content. We have chosen three basic diagram interaction styles and users can move between them at any time via editing modes buttons. Each approach provides users a different “look and feel” and each has certain advantages and limitations compared to the others. The options are:

- *Fully server-side* interaction processing, where all interactions (i.e. clicks on the diagram) are passed to the server-side web components and onto the Pounamu meta-tool for processing. After updating the shared copy of the source diagram, the diagram is redisplayed in the user's web browser. This style is the least like the thick-client tools as there is no “drag and drop” and any diagram click is posted to the server with high latency and full diagram refresh. However, it is the most portable across browsers and requires no browser (client-side) scripting support.

- *Buffered edits*, where a user's diagram edits are applied to their copy of the diagram only and cached on the server. These are subsequently sent on user request as a set of edits to the Pounamu server and applied as a unit on the shared copy of the source diagram. The diagram is then redisplayed in the user's browser. This isolates limited per-user changes and provides faster diagram refresh, at the risk of potential multi-user update conflicts.
- *Client-side scripting*, where some diagram interactions are handled by JavaScript in the client web browser. These include move, resize and deletion of diagram components using direct manipulation techniques. These are subsequently sent on user request as a set of edits to the Pounamu server and applied as a unit on the source diagram. This approach is the most similar to interacting with Pounamu thick-client diagrams but requires complex browser (client-side) scripting that currently limits cross-browser portability.

Examples using these techniques are outlined in the following three sub-sections. Following this we explore interaction with the same examples using the mobile phone interface and, to demonstrate versatility, a prototype 3D interface.

Fully Server-side Interaction Processing

Figure 4 shows opening and modification of an example UML diagram in Pounamu/Thin. The user first selects the UML model project to use (1). This loads the tool specification and model into the Pounamu server. The user can create or use an existing model project, and within this model project create and use diagrams of various types, in this case class, sequence, deployment and component diagrams. Each diagram type has a different set of symbols, editing syntax and semantics. In this example, a user opens an existing class diagram (2) and then adds a class element to the diagram by selecting the Add Entity button (3), specifying the kind of element to add to the diagram (a new Class in this case), and clicking where the new element is wanted, whereupon a blank class shape is displayed.

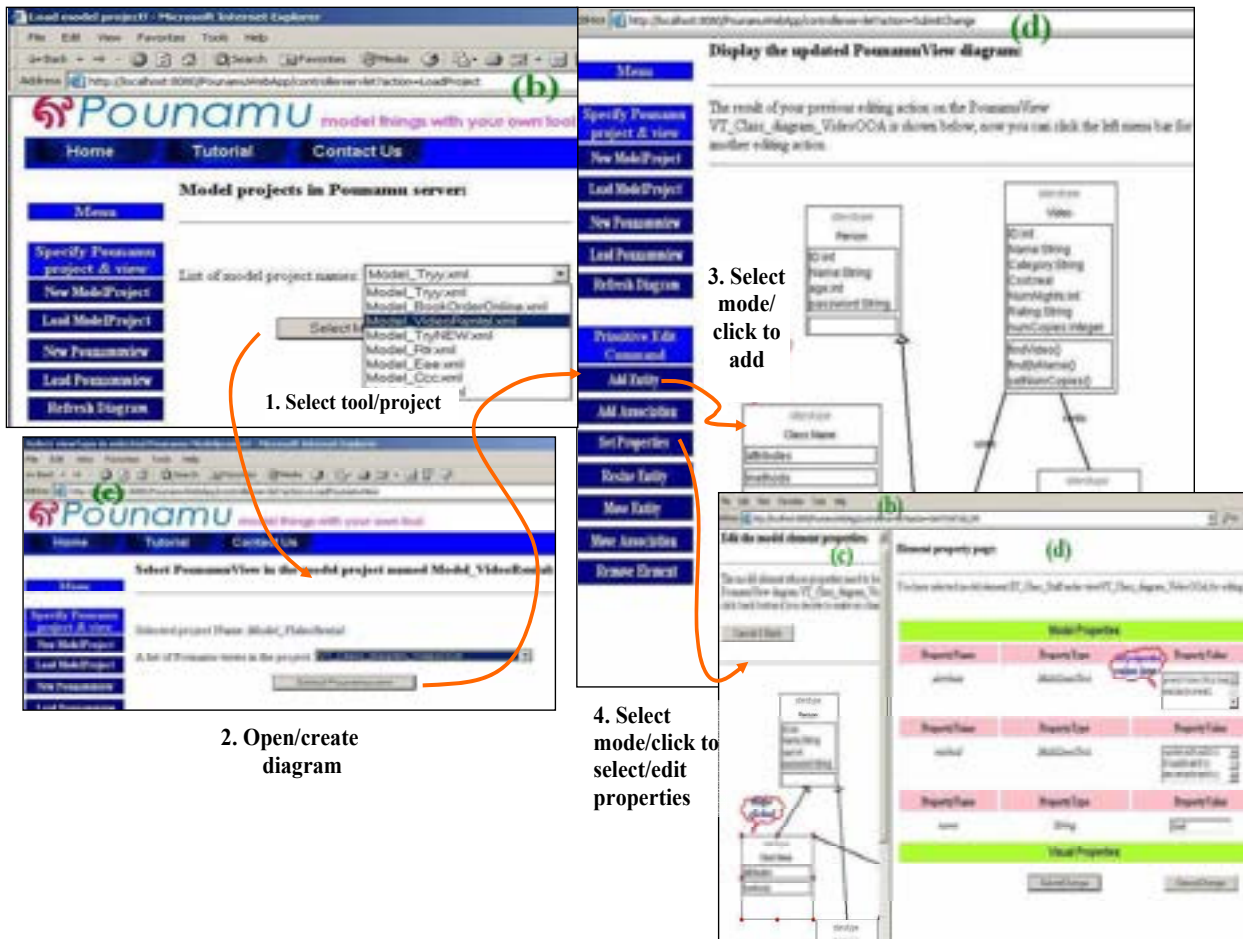


Figure 4. Interacting with the fully server-side processing client.

To set properties for this new element the user selects Set Properties, clicks on the shape and a property list for the currently selected element is shown (4). The values of properties in the list are editable with text fields, list boxes, radio buttons etc used to display and set values. Pounamu/Thin allows users to specify that they want this property list always visible in a frame to the right of their browser window for the currently selected shape, or to be hidden when not in use.

In this interaction mode for Pounamu/Thin, all editing operations are sent to the Pounamu/Thin web server and sent on to the shared Pounamu server for processing on the diagram data structure. All editing and semantic constraints for the diagram are implemented by the server. These might include automatic layout of shapes, automatic creation or deletion of shapes and connectors in response to edits, and semantic constraints on allowable editing actions.

As all diagram manipulations are done on the server-side, editing operations such as moving and resizing shapes require several interactions with the diagram via the browser window. Users firstly indicate they want to move an item by selecting Move or Resize Entity, as shown in Figure 5 (1). They select the element to manipulate by a mouse click (2). They then indicate the position to move it to (or place to resize selected corner to) by clicking in the location desired (3). The specified movement or resize operation is enacted by Pounamu/Thin and the modified diagram is redrawn. Similarly, connecting two shapes requires selection of the connector type, selection of the first shape to connect and then selection of the end shape to connect, each with a mouse click.

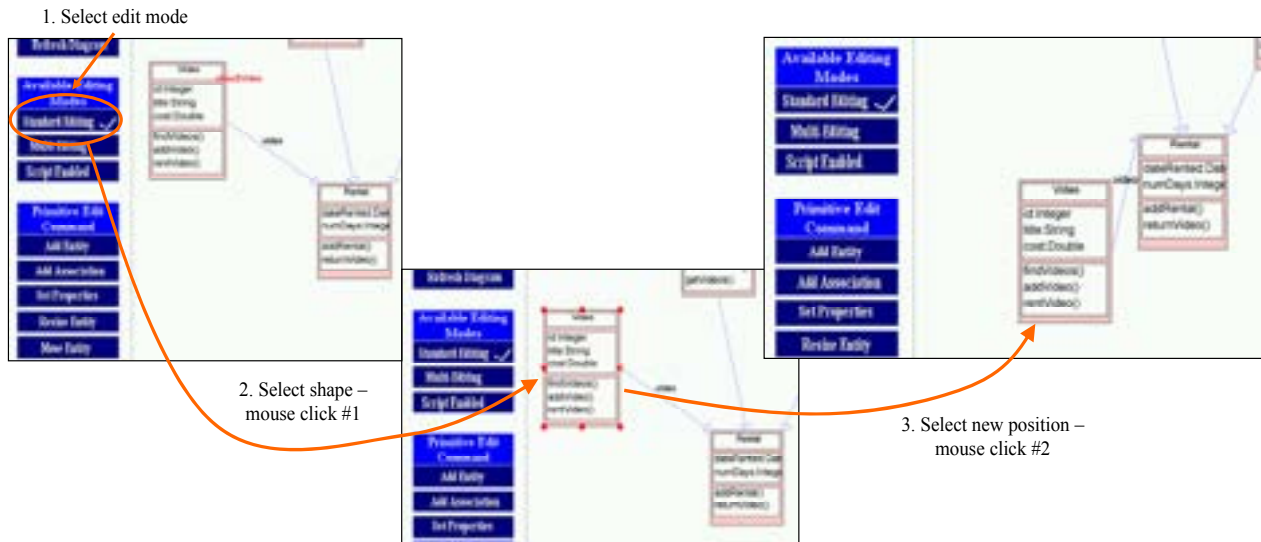


Figure 5. Example of moving a shape.

Collaborative diagram editing is supported with multiple users able to connect to the Pounamu/Thin web server and specify the same tool/project/diagram to view and edit. Concurrent editing of the same diagram element is prevented by the Pounamu/Thin server only allowing one user at a time to select a shape. Items being edited by another user are indicated as “locked by ...” awareness highlighting whenever the diagram is redisplayed.

Edit Buffering

From initial user feedback on using Pounamu/Thin we identified a common user desire to make a small number of “transactional edits” to their copy of the Pounamu diagram only. They will then commit these edits as one unit to the shared Pounamu diagram. Other users don’t see these “in progress” edits until all are committed.

To do this we developed a second interaction model that supports edit buffering. In this model each user has their own web server session with their own copy of the diagram. Figure 6 illustrates this. The user changes the current editing “style” to “Multi-Editing” (1), resulting in an extra set of menu items being shown at the bottom left of the browser window. The user then makes an edit to the diagram e.g. resizing the “Customer” class. Rather than send this edit to the Pounamu server, this user’s Pounamu/Thin indicates the edit is “buffered” i.e. remembered but not yet applied to the shared diagram, by a green dashed highlight (2). The user can throw away the change or retain it in the current edit buffer (3). Further edits can be made e.g. resize of the Staff class shape (4). Such “pending diagram updates” are highlighted in the diagram, as indicated by dashed outline of the Staff class shape.

When users have finished making a sequence of edits to the diagram, they select the Submit Buffered Editing operation. This sends the set of edits to the Pounamu server which updates the shared diagram data structures. The updated diagram is redisplayed in the browser (5). Updates made by other users are also shown in the re-rendered diagram. Some of these may be in conflict with the set of editing operations submitted by the user. In this case, the conflicts are resolved by the Pounamu server by using semantic constraints specified for the tool.

Client-side Scripting

Both of the previous interaction models require frequent browser-server interactions. The latency introduced can turn traditionally easy and highly interactive operations in thick-client diagramming tools into quite clumsy operations on the thin client. This particularly the case for moving and resizing diagram elements and adding or moving connections between shapes. To overcome this, a third interaction model we developed for Pounamu/Thin web clients uses browser (client-side) ECMA scripts to implement move, resize and connect operations in the browser plug-in.

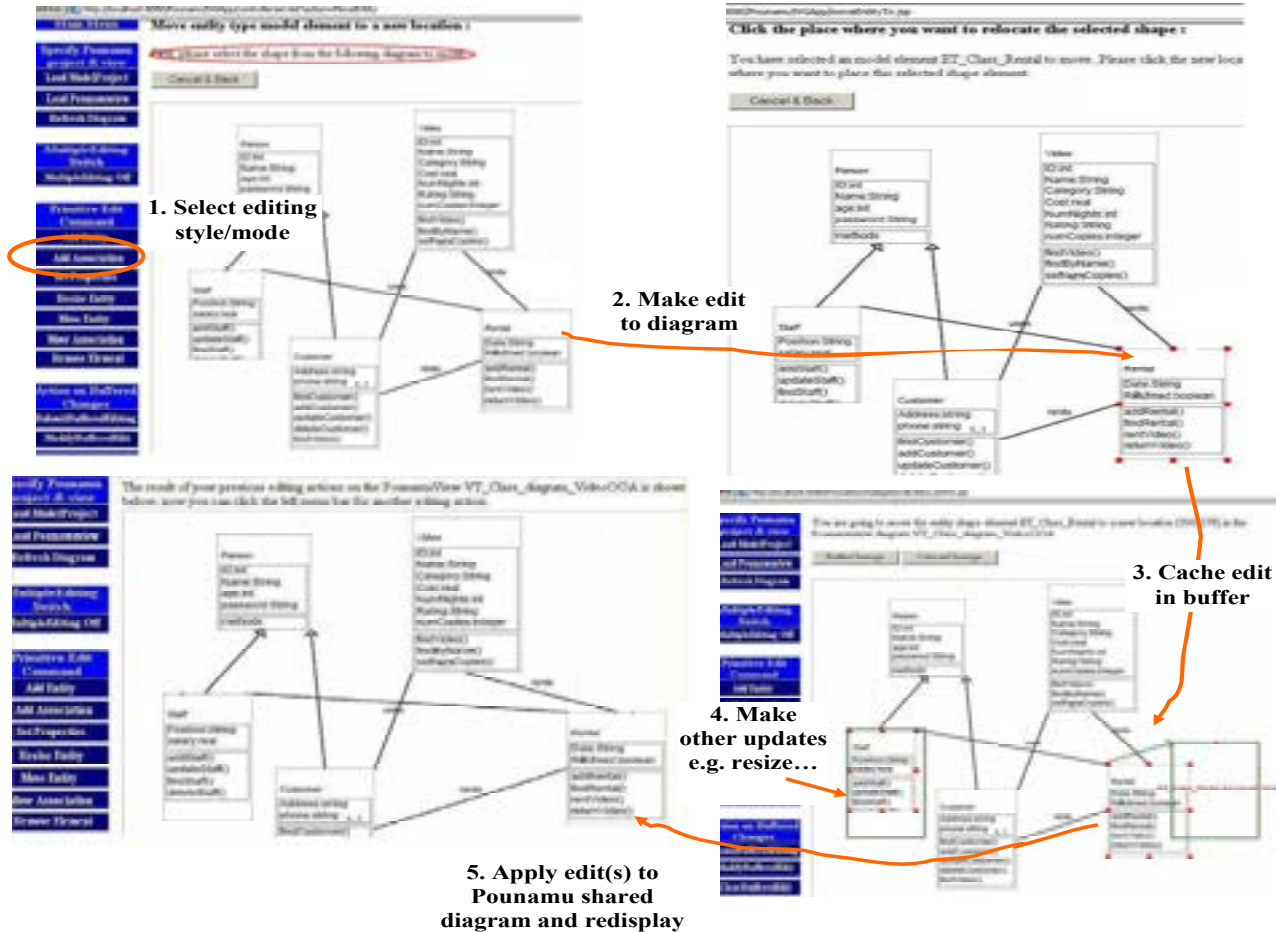


Figure 6. Examples of SVG-based thin-client editing with edit buffering.

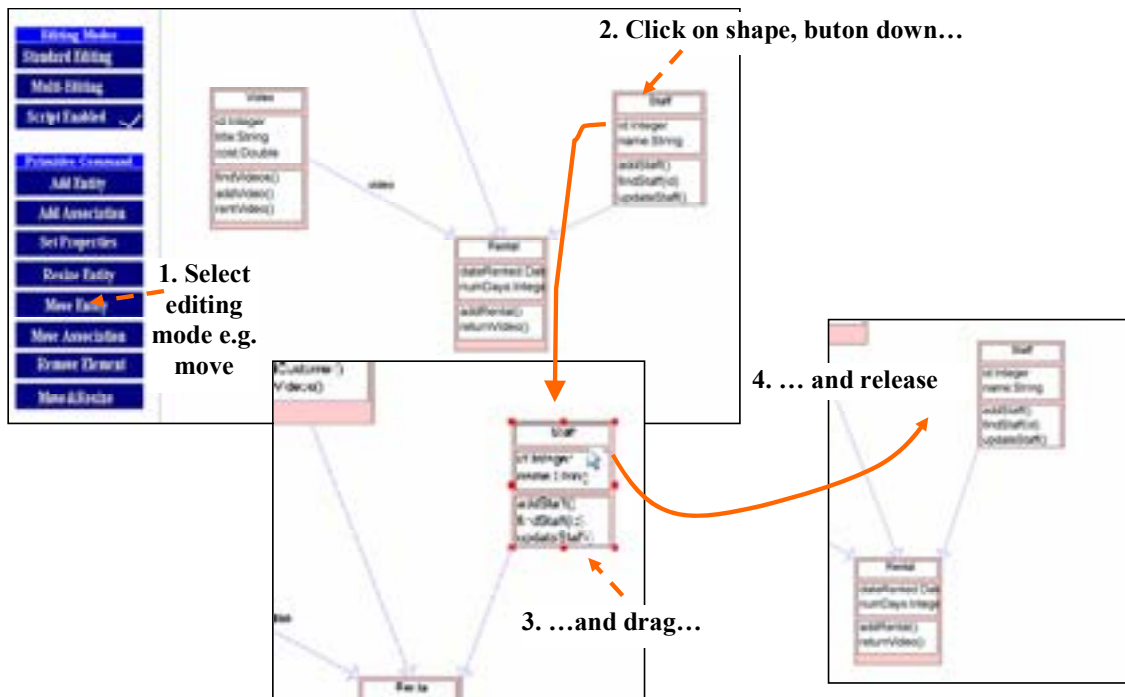


Figure 7. Example of client-side scripted based editing.

Figure 7 shows an example of diagram manipulation with client-side script editing. The user has chosen “Script Enabled” and the diagram has been reloaded, with ECMA script sent to the browser to configure the SVG plug-in. When the user selects a Move operation (1), interactions with the diagram are detected and handled by this client-side ECMA script instead of being posted to the web servlet as previously described. For a move operation on the diagram the user clicks on the shape to move, holding the mouse button down (2), drags the mouse to the desired new shape location (3), and then releases the mouse button (4). This set of interactions is just the same as with conventional thick-client direct manipulation tools like Pounamu. The diagram elements are moved, in this example the Staff class shape and its association connector line to the Rental class shape. Further move operations can be done in the browser by the user. These are not reflected in the shared Pounamu diagram until the user selects another operation e.g. Add Entity, Resize, Connect Entity etc.

Pounamu/Thin on a Mobile Phone Device

A number of industry partners we have been working with have indicated a desire for rich diagram-based user interfaces on PDAs and mobile phones for a range of application domains. To experiment with the feasibility of such interfaces we have developed prototype support for Pounamu/Thin diagrams to be viewed and edited using mobile phones and PDAs. Figure 8 (a) is the post logon Pounamu/Thin screen, showing a list of viewing and editing options. After selecting a diagram to display, an initial overview of it is generated, such as the class diagram overview shown in (b). Views may appear differently on different mobile devices as the overview is generated by proportionally shrinking original diagrams to fit them onto the small screens of mobile devices. The proportion is calculated according to the size of the selected Pounamu view to display and the screen size of the client device requesting a page. Detailed diagram contents are filtered out in overview mode because they are hard to see. Users can press direction keys to select diagram shapes and connectors. The status bar underneath displays certain details of the currently selected item, in this case the name and type of the item.

As web browsers typically run on PCs with large screens Pounamu/Thin diagrams displayed in web browsers are close to the fidelity of thick-client versions. With client-side scripting support user interaction with a web browser-based Pounamu/Thin diagram can be a similar experience to that of thick-client diagram tool interaction. However, PDAs and mobile phones are much more limited in their display and interaction capabilities. To help address the rendering and resolution limitations of most mobile devices, we developed a multi-level rendering and zooming capability. This facility allows users to define multiple representations for diagram elements at different levels of detail. Each diagram element can be separately selected and zoomed among the multiple levels. Automatic focus and context zooming of elements is supported as users navigate a large view. We use several navigation approaches to improve user interaction with diagramming tools on mobile devices. Button-panning let users quickly move around in a big diagram using mobile phone buttons. A floating zooming window allows users to easily and quickly pan to interesting areas from an overview of a large diagram.

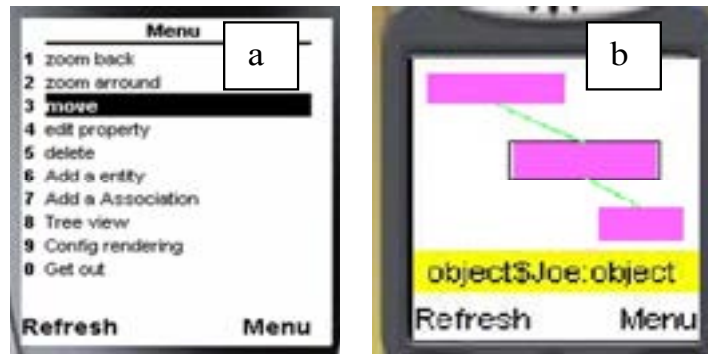


Figure 8. (a) main menu of Pounamu/Thin mobile tools; (b) overview of class diagram view on a colour phone.

Figure 9 shows an example of this multi-level zooming and pan navigation. To zoom in/out diagram shapes to various detail levels, firstly a user selects a diagram shape from the model view, and then presses a pre-defined hot key to zoom into a particular level. In Figure 9 (a), all diagram elements are at “zoom level 1”, providing a high level overview of the diagram but no details for each element. In Figure 9 (b), one object entity is zoomed in level 2, and two class entities are zoomed in level 3. The user can select individual elements and zoom them in or out, or can enable an auto-zoom facility that magnifies the selected item and its immediate neighbours while de-magnifying those further away. This forms a basic distortion-oriented display. As each Pounamu diagram type can have different element types and connectivity, different view types have different auto-zoom behaviours. To use pan navigation, a user selects pan navigation from the options menu or a hot-key button. They can zoom out to the top-level overview and a floating window indicates the current zooming area in the model view, as shown in Figure 9 (c). The floating window can be moved across the diagram using mobile phone keys and selecting a hot-key magnifies elements in the selected zooming area.

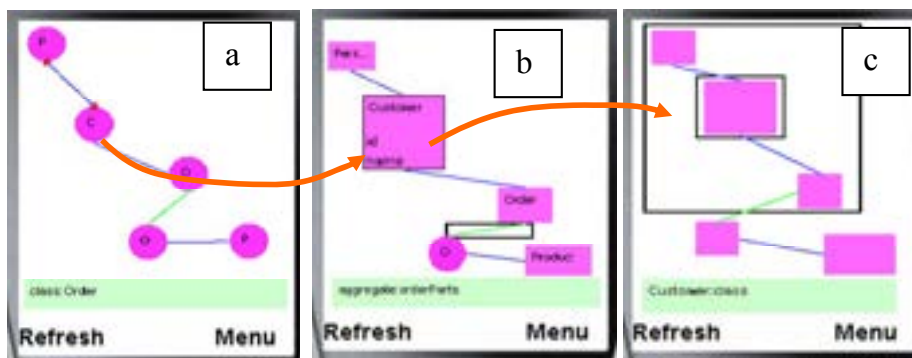


Figure 9. (a) zoomed in level 1; (b) shapes zoomed to different levels; (c) pan navigation to selected area.

Adding, moving, deleting, and editing items in a diagram require similar interactions through the generated Pounamu/Thin mobile device user interfaces. Firstly a user selects an item to manipulate, and then chooses a menu command or a hot-key indicating the action they want to perform on that item. For example, to move a shape, the user selects a shape, then repositions the shape using the mobile device’s direction keys, and confirms the new position by pressing e.g. the Select hot-key on the device. Items are added to a place on the screen by moving the pan selection floating window and narrowing it down to an unoccupied space with hot-keys.

To edit properties of an item, the user selects the item, as shown in Figure 10 (a), then selects a hot-key or menu option to edit its properties. A list of the element’s properties is displayed which the user edits with conventional mobile device interactions, as shown in Figure 10 (b). The user then selects the OK menu to submit the values back to server. After processing the user request, the server sends back an updated view.

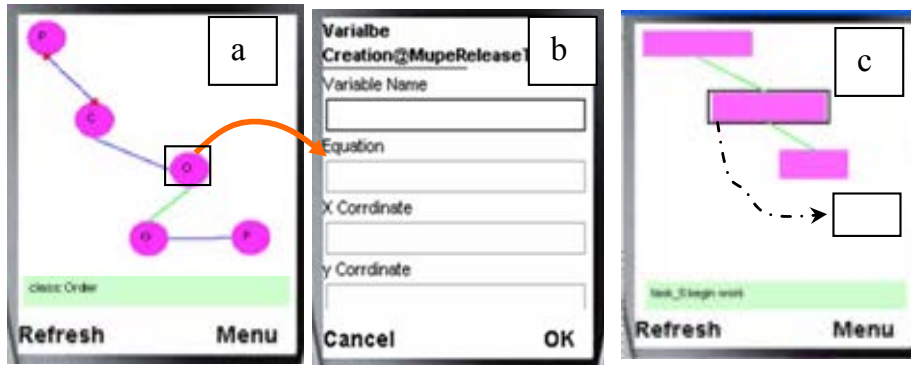


Figure 10. (a) select an element; (b) edit the element’s properties; (c) and moving the element in Gantt chart.

One issue we repeatedly encountered when building multi-device user interfaces is the inability of generated interfaces to suit all potential users. In particular, when providing multi-level zooming capability on a mobile-hosted diagram, we have found that different users want different element renderings for different zoom levels. This, along with our desire to experiment with different mobile rendering approaches led us to support such user preferences by using a multi-user runtime Pounamu/Thin application configuration facility. Users can specify different diagram content, zoom, and navigation configurations via their mobile device. These individual configurations are stored in XML format in the Pounamu/Thin Server. At run-time the system obtains model information to display from the Pounamu application server and generates diagram views according to each user’s configuration. Default user configurations are generated and these may be tailored to enhance users’ display and interaction.

Figure 11 (a) shows an example of tool configuration via mobile user interfaces at runtime to specify different element appearance at different zoom levels. To configure diagram representations, a user selects the type of entities that they want to specify the appearance of and then the menu entry “Config rendering”. The mobile server finds the user’s Pounamu tool configuration in xml format and generates configuration user interfaces for it and the user selects the zoom level to specify a representation for. A diagram representation currently can be specified by using color, size, icon shape, and properties to show. Figure 11 (b) shows the result of a user configuring a UML class diagram view in which all diagrams are zoomed in level 1. Diagram representations are different from the one shown in Figure 5 (a) as different shapes have been selected for the icon representations.

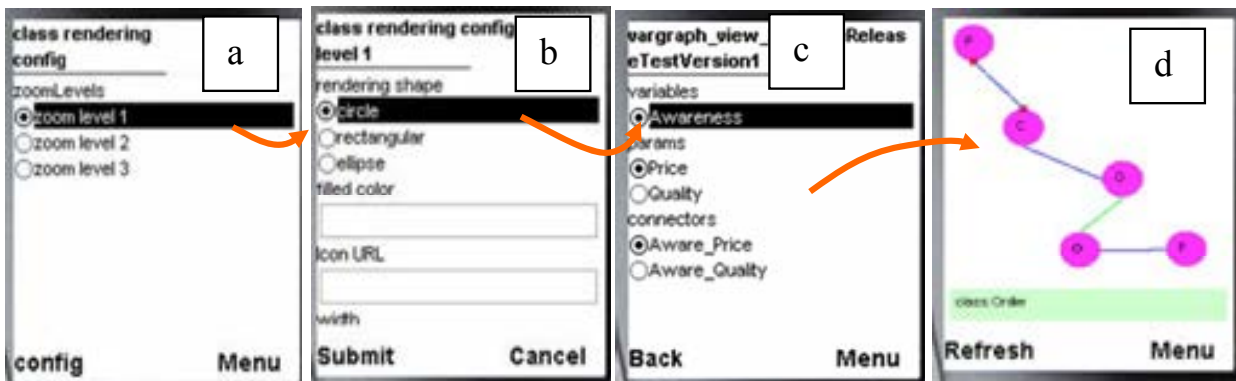


Figure 11. (a-c) user configuring class diagram representations in multi-level; (d) the class diagram with all items zoomed to level 1 as specified by users.

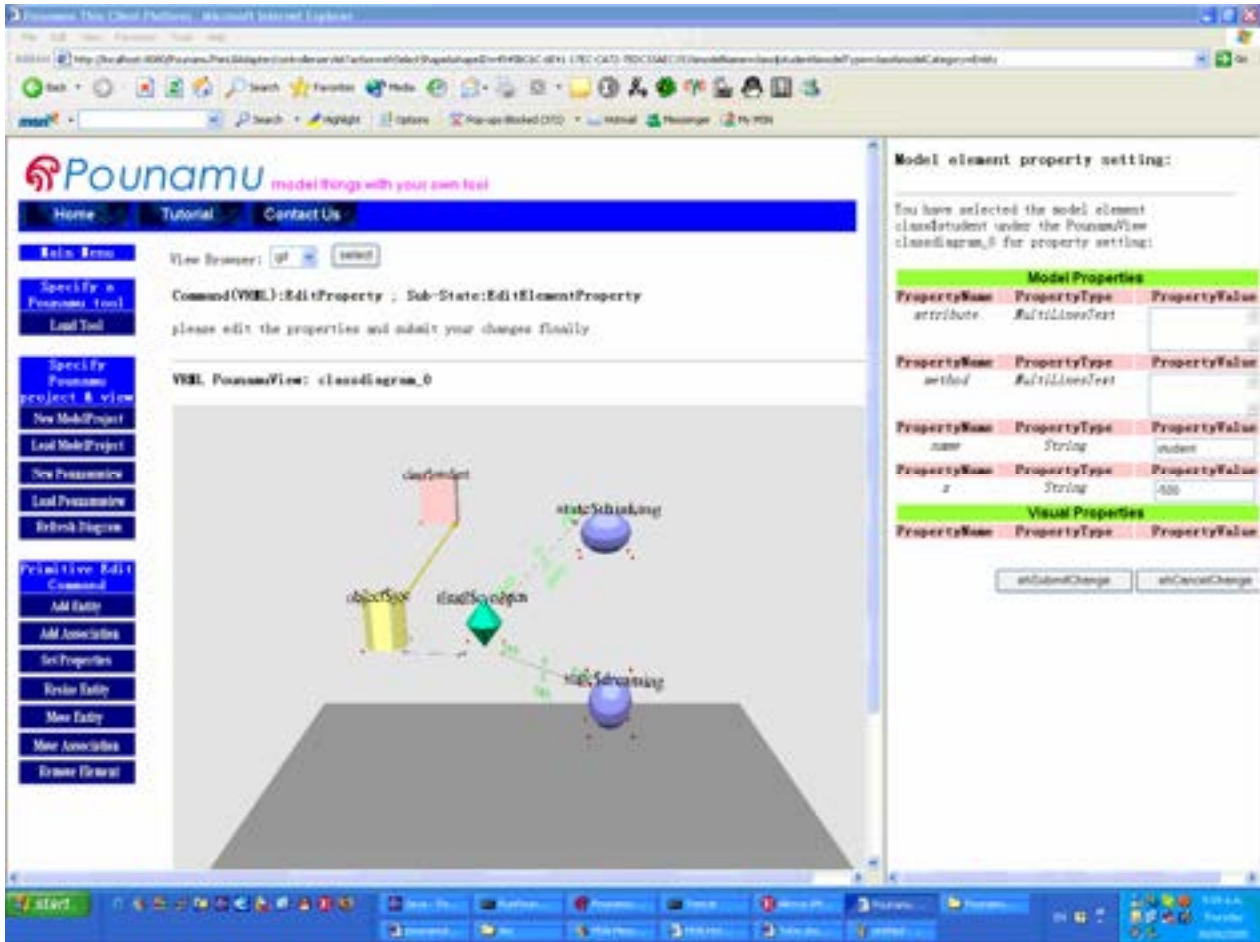


Figure 12. 3D diagram viewing and editing.

3D Views for Pounamu/Thin

To demonstrate versatility of our Pounamu/Thin architecture, and motivated by emerging 3D environments such as Project Looking Glass [43], we have developed a further extension of Pounamu/Thin allowing diagrams to be converted into 3D representations and rendered and edited in a web browser. This is useful in situations where complex diagrams may benefit from a third dimension to help handle complexity and layout. Figure 12 shows an example of a 3D class diagram where the elements have been converted to 3D symbols (cylinders, spheres and cuboids), with selected labels rendered. The user has selected the student class in the 3D diagram to edit its properties, shown on the right hand side. The user may use the 3D plug-in capabilities to zoom in and out, pan and rotate the 3D representation. Client-side scripting is used to support adding, moving, resizing and deleting of diagram shapes as in the 2D web-based client-side editing example outlined previously. Using this extension, we are experimenting with a range of 3D metaphors for visualising model information. These involve configuring additional mappings from underlying Pounamu model elements to generate “the 3rd dimension” from model element attribute information.

POUNAMU/THIN ARCHITECTURE

Our Pounamu meta-tool was originally developed to provide flexible, interactive diagramming tool specification. It was assumed that all of the specified diagramming tools would provide a conventional thick-client user interface to tool users, by interpreting these design tool specifications. However, we increasingly found our end user base wanted to deploy their tools using thin client interfaces. One approach was to build a stand-alone web application that could import Pounamu tool specifications and realise these thin-client editors. However this would mean repeating much of the complex management support already developed for Pounamu e.g. data persistency to XML files and databases; model and view data structures and consistency management; and diagram rendering and editing support.

We pursued an alternative approach developing an optional thin-client diagramming extension to Pounamu. This is a set of web components developed independently of the Pounamu meta-tool itself but that make use of existing Pounamu APIs. The thick-client Pounamu application is thus repurposed to act as an “application server” reusing all of its

standard data management, persistency and diagram editing support. The web components communicate with Pounamu to extract diagram information to render and to update diagram content. Our aim in doing this was to support thin-client diagram viewing and editing, but without the need to change or replicate any code in Pounamu itself, and still using the same tool specifications as thick-client tools. Figure 13 illustrates this thin-client extension that we call Pounamu/Thin.

Initially a tool designer specifies a diagramming tool using the thick-client tool design facilities of our original Pounamu meta-tool. These tool specifications are saved to an XML-based tool repository for use by the Pounamu tool interpreter, which originally provided only thick-client diagramming facilities. To run the tool, Pounamu and Pounamu/Thin web components are deployed and the tool specifications loaded by Pounamu and interpreted. The Pounamu/Thin web components interact with Pounamu via a remote RMI-based API. As Pounamu is a “live” meta-tool, the tool specifications such as diagram element appearance, meta-model entities and attributes, view definitions and event handlers, can be changed while the tool is in use. This is currently done using the thick-client tool designer but future extensions might use a thin-client meta-tool designer via the Pounamu/Thin architecture itself.

Having deployed a tool users can then access the tools diagrams generated by the Pounamu/Thin web components via conventional web browsers, using GIF based rendering by default. SVG-based rendering of diagrams is supported but this also requires an SVG plug-in in the browser. The advantage of SVG diagrams is that they are rendered on the client-side resulting in higher resolution than GIF-based diagrams. In addition, SVG rendered diagrams also support client-side browser scripting for more interactive diagram manipulation, including drag-and-drop direct manipulation for moving and resizing of content. Similarly 3D viewing of diagrams in the web browser requires a VRML plug-in, which also performs client side rendering and script-based client side interaction.

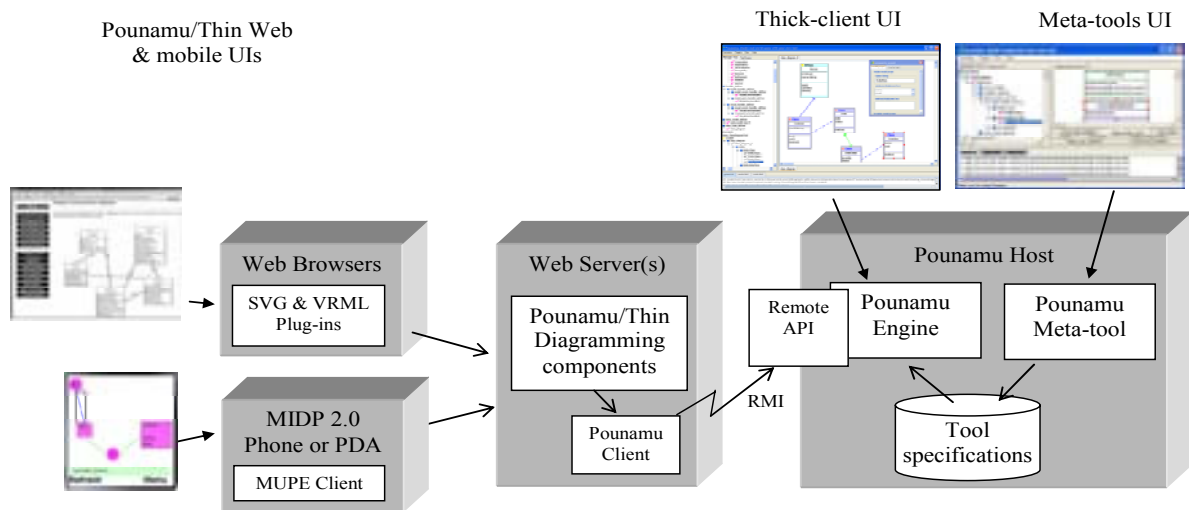


Figure 13. High-level Pounamu/Thin software architecture.

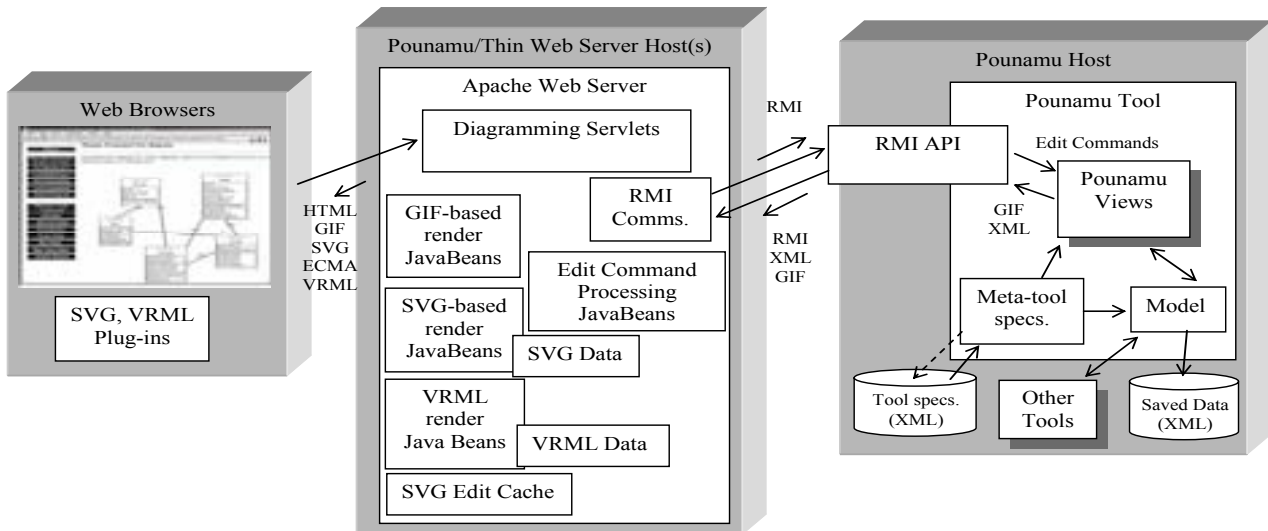


Figure 14. Architecture of Pounamu/Thin.

For deployment of Pounamu/Thin interfaces on mobile devices we chose to use Nokia's Multi-User Publishing Environment (MUPE) mobile client technology. MUPE is a mobile thin client technology developed by Nokia consisting of a MUPE server and MUPE client browser. The MUPE server responds to client requests by generating and sending back pages in XML, which can be browsed in a MUPE client. MUPE clients are relatively lightweight and are based on Java MIDP, which is enabled on most newer mobile devices. By using MUPE, diagrammatic user interfaces generated by our system can be easily deployed on a wide range of MIDP-compatible devices. Initially we implemented a "Pounamu/Mobile" server as a set of components hosted on a MUPE server. More recently we have refactored these components to use the same approach as for the web browser version with translation to and from the MUPE XML mark-up language.

DESIGN

Core Components

Figure 14 illustrates the key architectural components of Pounamu/Thin's web browser support. The Pounamu tool runs on a host and behaves as the application server/database manager. Tool specifications are loaded from XML repositories and the design tool configured from these. Multiple views of a model are provided by Pounamu, with the diagram views the point of interaction for users. View data can be converted into an XML format or a GIF format.

Pounamu uses a Command pattern approach with a set of "Command objects" to represent edit operations being made to a view (or model) elements. These have execute(), undo(), redo() methods which when invoked carry out the specified modification of Pounamu data structure state and re-render affected views. Command objects include AddShapeCommand, AddConnectorCommand, MoveShapeCommand, DeleteShapeCommand and SetPropertyCommand. These objects can be created and sent to a Pounamu view programmatically via an API in order to modify it. The API, which has RMI and SOAP variants, thus enables Pounamu views to be created and modified remotely.

Pounamu/Thin is a set of web components hosted by an Apache web server. A set of Java Servlets provides the diagramming interface, and includes user login, view manipulation (create, display etc), diagram rendering and manipulation, and shape or connector property editing facilities. A set of JavaBeans provides support infrastructure, including GIF, SVG, and VRML based diagram rendering, Pounamu Command object construction for diagram editing, a copy of the SVG and VRML diagram data per user, and SVG and VRML edit command cache per user.

Users interact with the Pounamu/Thin diagramming tools via standard web browsers. However, if SVG or VRML diagram rendering is desired, an SVG or VRML plug-in must be present in the user's browser. Multiple users can view and edit the same diagram simultaneously. The web component servlets provide co-ordination via element locking and sequencing of Command message sending to the single Pounamu tool acting as the shared application server.

We chose to use a set of servlets to produce GIF, SVG, MUPE or VRML content for the browser with optional client-side scripting of SVG and VRML plug-ins to provide a degree of client-side drag-and-drop interaction. This is in contrast to using a Java Applet or Active X control in the browser with a similar editing interface to the Pounamu thick-client application. Our choice of using servlets that generate content to be rendered by the browser rather than an applet or Active X control was motivated by several factors: cross-browser compatibility – our own and others experiences with both Active X controls and Java applets for such rich client interfaces in previous work (Graham et al 1999, Evans and Rogers, 1997) demonstrate major browser configuration problems are common occurrences. In addition, the use of servlets and translation to/from browser content and requests allowed us to use a common server-side architecture, rendering components and editing components across a range of diagram representations. However, our approach does not preclude using a Java applet or Active X control as Pounamu/Thin options in the future - both can communicate with the Pounamu/Thin servlets and request XML for rendering diagrams and send Pounamu commands as XML messages to modify diagrams.

Web Browser Interaction Models

As illustrated previously Pounamu/Thin provides several different 2D web browser diagram interaction models for users. These range from a full server-side processing model where images are rendered in a browser and all user interaction sent to the server, to a highly interactive client-side scripting approach. A fully server-side processing model is used for GIF image diagrams and can also be used for SVG diagrams. This approach is illustrated in Figure 15. Here, all interactions with the diagram are sent directly to the web server components (1). These generate an appropriate diagram update message (a Command) and immediately pass the Command onto the Pounamu server (2). These Commands are processed by the server and updates are made to the shared diagram. The server updates the shared diagram (3), and associated model elements, resulting in updated diagram and model. Periodically the model and diagram data structures are saved to XML files on the server (4). The Pounamu/Thin servlet then requests the updated

diagram contents (5) and uses this to re-generate HTML, GIF and/or SVG content as appropriate for redisplay in the web browser (6).

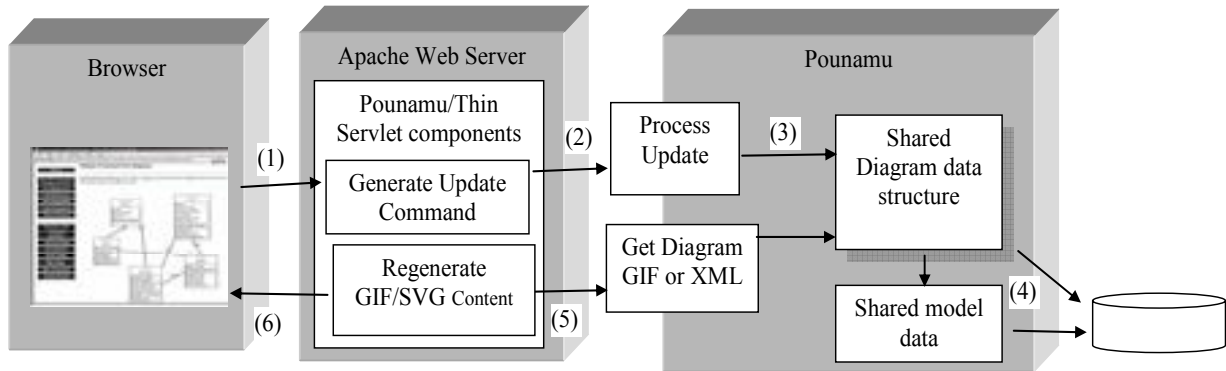


Figure 15. Fully-server side processing model.

The web server-side edit buffering approach is illustrated in Figure 16. In this approach the Pounamu diagram contents are fetched from the shared Pounamu server (1) and each user's web server session caches a copy of the diagram. User interactions with the diagram are processed by the Pounamu/Thin servlets (2) which update the user's copy of the SVG diagram contents in the servlet (3). The updated diagram is then redisplayed (4). At some point the user will ask for their edits to be "merged" into the shared diagram held by the Pounamu server (5), which processes the updates as a transaction on the shared diagram (6), also resulting in shared model data structure updates and save of diagram/model state (7). The servlet will then fetch the contents of the diagram back from Pounamu, as it may contain the effects of edits made by other users as well as their own (8). The updated diagram will then be converted back into SVG or VRML and redisplayed. This approach is only available for SVG and VRML-format diagrams as GIF images are generated on the Pounamu server which does not support edit buffering.

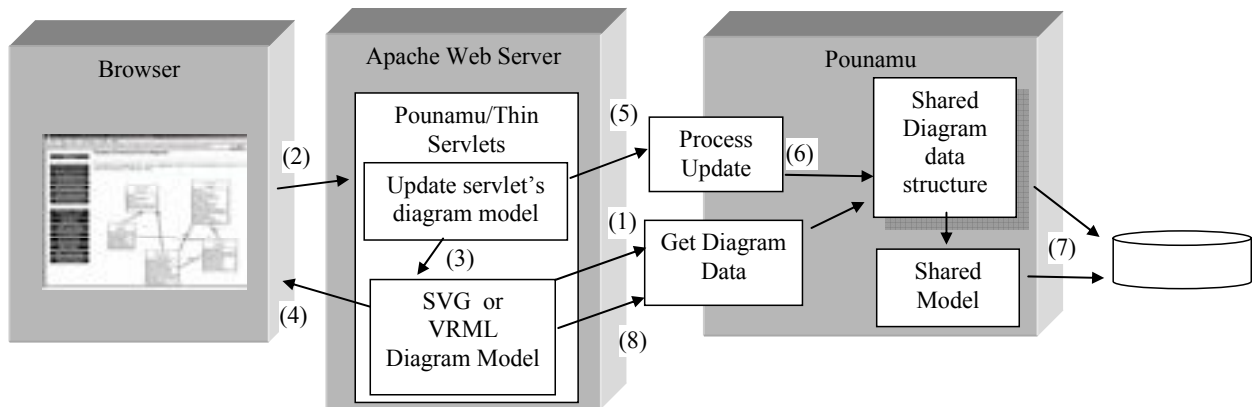


Figure 16. Edit buffering in servlet model.

The client-side scripting approach is illustrated in Figure 17, again available for SVG and VRML format diagrams. In this approach the Pounamu diagram contents are fetched from the shared Pounamu server (1) and cached for each user session in the servlets. In addition to HTML, SVG and VRML content, the servlet generates ECMA scripts for the SVG plug-in and JavaScript for the VRML plug-in (2). These scripts are passed with the HTML, SVG and/or VRML content to the web browser. These scripts program the browser plug-in to provide the Pounamu/Thin client-side editing behaviour (3). When a user moves, resizes or connects shapes, the edits are implemented in the browser and not sent to the web servlet (4). After several edits users can indicate they want the client-side edits reflected in the shared diagram and a summary of the edits is sent to the servlet (5). Appropriate update messages are sent to the shared Pounamu server (6) where the shared diagram is updated (7), and the shared model updated and diagram/model data structure state periodically saved (8). The updated Pounamu diagram content is refetched by the servlet as XML data (9) and the SVG or VRML regenerated by the servlet to reflect both the client-side updates and any other user updates.

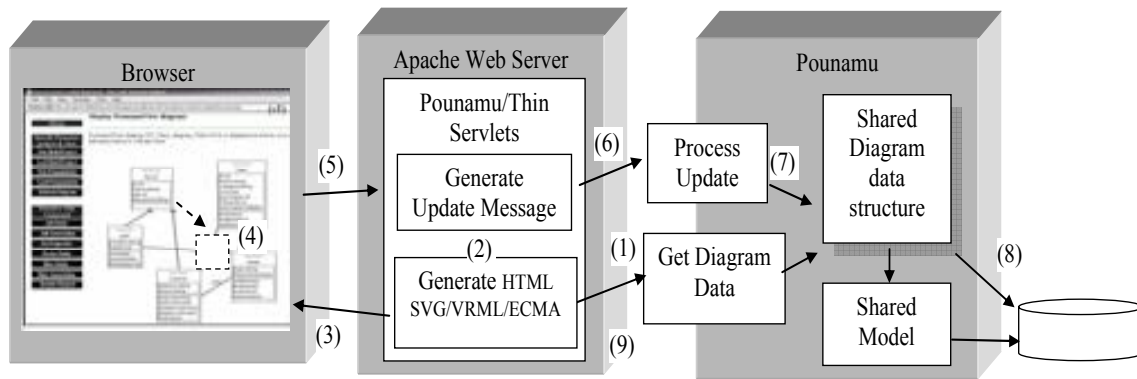


Figure 17. Client-side scripting interaction model.

We used a similar approach for the VRML-based 3D rendering of Pounamu/Thin diagrams. Pure server-side processing of user requests is too cumbersome when trying to navigate and modify a rendering in 3D in a VRML plug-in. Instead we used a client-side scripting approach very similar to the one used for SVG. If a 3D rendering of a Pounamu diagram is requested, the Pounamu/Thin components generating the VRML content include a set of ECMA scripts which program the VRML plug-in to support interaction and modification of the VRML content within the browser. This includes support for double-clicking on items and adding, moving, resizing and deleting them. ECMA script directly modifies of the VRML data structure within the browser plug-in and uses asynchronous posting of update requests to the Pounamu/Thin server.

Likewise, displaying and interacting with diagrams on a mobile device can be done using fully server-side processing as with GIF diagrams in a PC web browser. However this proved slow and cumbersome due to the high latency and slow speed of mobile device networks. Instead we used a Java-based MIDP2.0 plug-in on the client handset to provide client-side scripting, allowing much of the diagram navigation, zooming and interaction to be carried out on the handset. This greatly reduces network traffic between the handset and Pounamu/Thin server. We used the Nokia MUPE framework to provide a set of Pounamu/Thin server components specifically targeted at providing the mobile device interfaces. While this approach is applet-based, in contrast to the web browser based approaches of the other Pounamu/Thin applications, this is justified by the lightweight and “standardised” nature of the MUPE client in comparison with typical web browsers available on Mobile phones and PDAs.

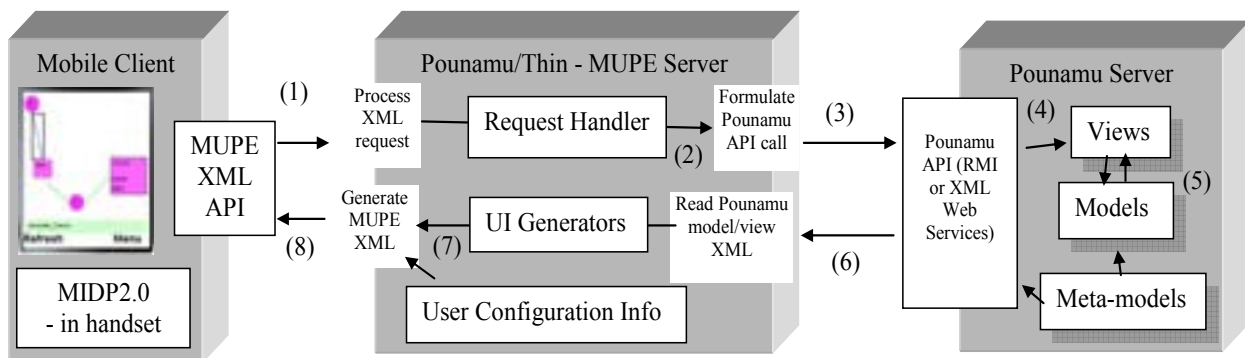


Figure 18. Basic components in Pounamu/Thin for mobile device clients.

Figure 18 describes diagram editing in a Pounamu/Thin mobile client. If a user asks for an element to be deleted, a MUPE XML event message is sent by the MUPE client on the device to the MUPE server (1). A Pounamu/Thin request handler determines the diagram update required (2) and formulates a Pounamu API call to action the diagram update (3). This results in the element being deleted from the Pounamu view (4), also deleting the model element and impacting other views of the deleted element (5). Once the Pounamu server acknowledges the update has successfully occurred, Pounamu/Thin re-reads the updated view’s Pounamu XML from the Pounamu server (6). It then synthesises a new view in the MUPE XML format, using the user preferences and Pounamu view XML to do this (7). The MUPE client is returned a new page to display, the updated diagram content (8). Various optimisations are possible, including caching the MUPE XML in the MUPE server and only updating affected portions. However we have found the response time when re-generating the whole diagram content adequate and this simplifies the process considerably.

Rendering Transformations

In order to produce content for a thin-client device, whether web browser, mobile device, or 3D image, Pounamu diagram content must be transformed from its XML format into suitable thin-client device content (GIF, SVG, VRML, MUPE XML). In addition, if client-side scripting is used, and it must be for mobile and 3D content, the payload for the target device must include suitable scripting code as well as content to render. Figure 19 shows a sequence diagram outlining the generic transformation process when a diagram is requested by a client device. The process is slightly different for each kind of content and is currently implemented by different Pounamu/Thin server-side web components, but, as can be seen, the general pattern of execution is the same.

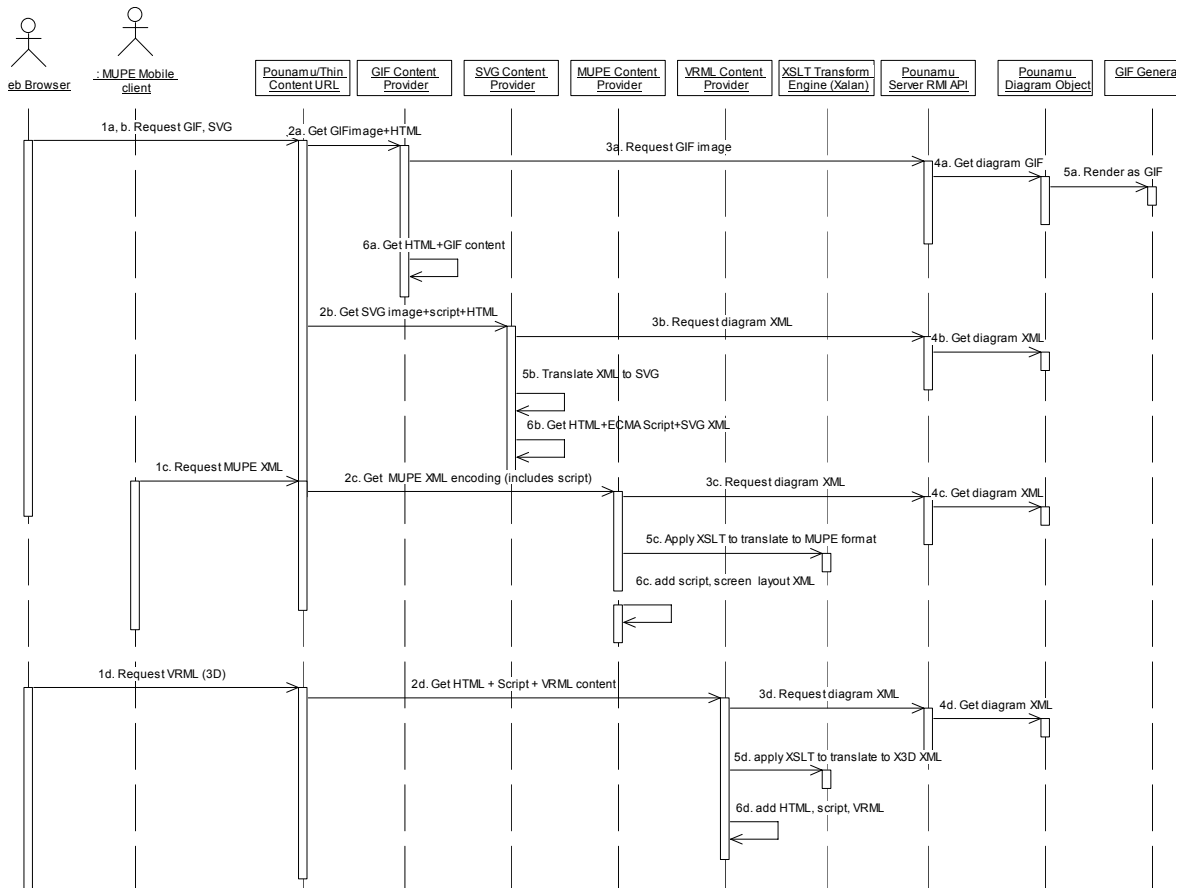


Figure 19. Outline of the Pounamu/Thin XML transformation process.

All requests are sent to a Pounamu/Thin URL with arguments to indicate the kind of content to provide (Figure 19 1 a-d). When requesting a GIF image the GIF Content Provider servlet is used to generate HTML and the GIF image to return to the client web browser (2a). This requests a GIF image for the requested diagram via the Pounamu server API (3a). Unlike other content the GIF image currently must be generated by a plug-in to the shared Pounamu application server; we use a separate Thread to generate these images and re-render a copy of the diagram to a binary array in a GIF format (4a, 5a). The GIF Content Provider servlet then returns HTML markup providing the title, menus, frames and the GIF image to the client browser. SVG content requests may include scripting (if client-side scripting mode is requested by the user) as well as HTML and the SVG image (2b). The SVG Content Provider requests the Pounamu diagram as an XML document (3b, 4b) and transforms this into an SVG image by traversing the diagram XML in a Document Object Model (DOM) (5b). We currently implement this transformation as optimised Java code over the DOM structure. In future it could use an XSLT or similar transformation engine as do the MUPE and VRML content production.

Mobile devices are currently provided a MUPE XML document as their content to render and are currently required to have Java MIDP2.0 and the MUPE client handset plug-in installed (1c). The MUPE Content provider requests the Pounamu diagram XML document (3c, 4c) and then applies an XSLT transform to the document to convert it from the Pounamu's diagram XML format into MUPE diagram rendering XML format (5c). The MUPE Content Provider then adds this diagram content XML inside a new XML document incorporating screen layout and handset scripting XML (6c), producing the final MUPE XML document to be returned to the mobile device.

Requests for 3D rendering of Pounamu diagrams (1d) are processed by a VRML Content Provider (2d) which requests the Pounamu diagram XML (3d, 4d) and transforms this into X3D, an XML document structure (5d). The VRML Content Provider servlet could equally produce a VRML document (non-XML) but we chose to use X3D, the newer XML-based representation for VRML content. HTML and ECMA script content is generated by the VRML Content Provider servlet and the X3D document included by the HTML to provide the 3D representation and associated scripting and HTML menu and frames content for the client web browser (6d).

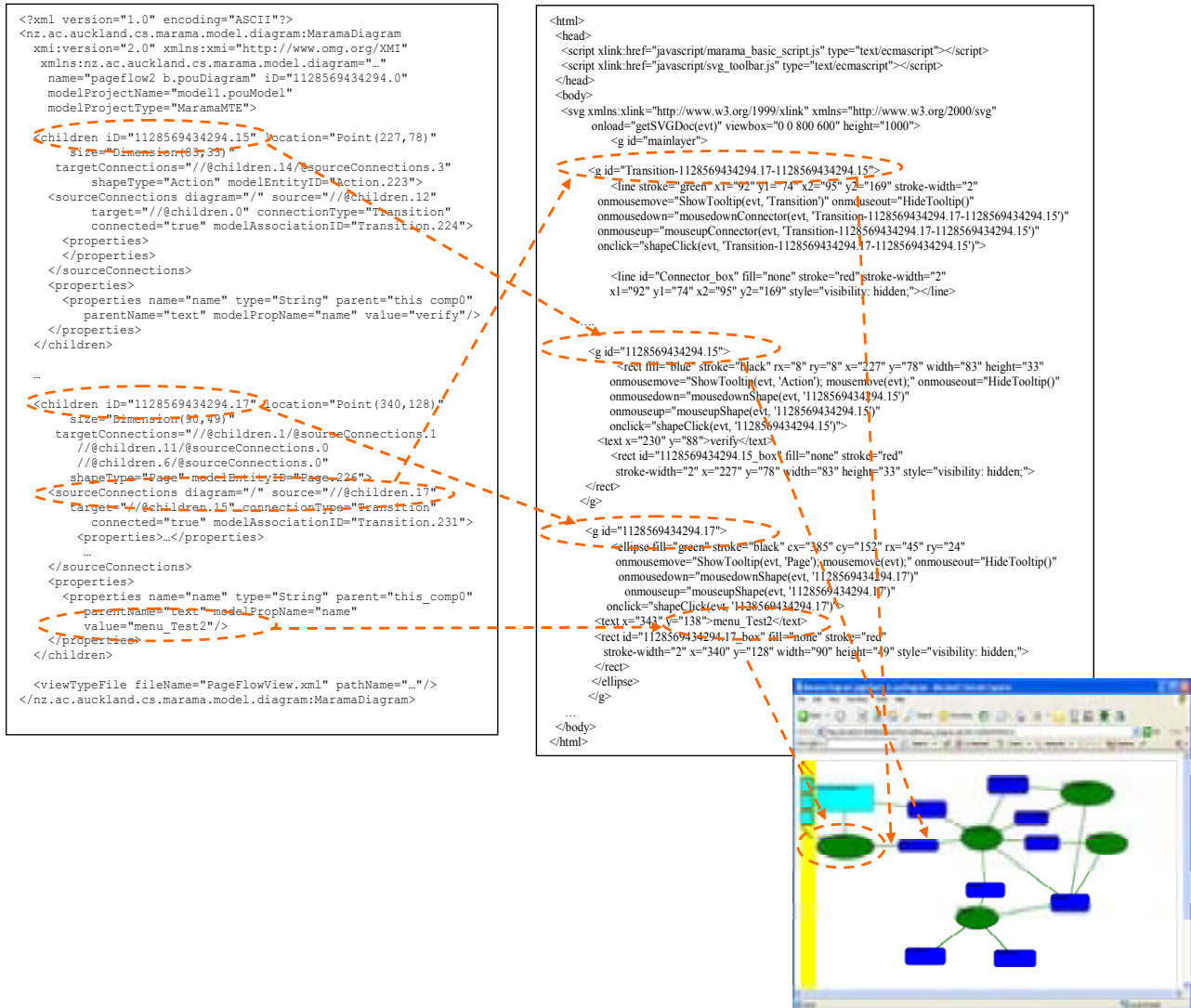


Figure 20. Example of Pounamu diagram XML transformation to SVG.

Not shown in Figure 19 are caching and threading structures used. Rather than use the Pounamu modelling tool API calls to get diagram XML the Pounamu/Thin web server components can share a cached copy. This allows multiple requests for the same diagram content from different users' to be generated once when the content is changed in the Pounamu tool server and served to each user without re-generating the diagram XML. Similarly some client plug-ins cache a copy of the diagram data structure on the client device e.g. the edit buffering and client-side scripting versions of the SVG content, the MUPE mobile client content and the VRML plug-in content. Client-side scripts for each of these target devices can modify the diagram data structure (held as SVG, MUPE or X3D format XML documents in the client-side browser plug-in) without communicating immediately with the Pounamu/Thin web server. This greatly improves response time for e.g. drag-and-drop editing for the user. Threading is used on the Pounamu/Thin server to provide each client their own copy of all Pounamu/Thin servlet objects. Synchronisation points are on the cache update method for cached Pounamu diagram XML documents and on all of the Pounamu server RMI API calls, eliminating concurrent operations on these.

Figure 20 shows an example of the transformation between a Pounamu diagram representation as an XML document (left) and a target SVG XML document (right) for rendering by the client web browser SVG plug-in. Several

correspondences between Pounamu diagram XML structures and SVG document items are highlighted. In addition, the SVG Content Provider generates HTML to contain the SVG document and includes several JavaScript files providing client-side scripting support for drag-and-drop move, resize and delete implemented by the client web browser SVG plug-in. The resulting rendering of the diagram in the browser is also shown (bottom).

In order to update shared Pounamu server information, all modifications to diagrams by user interaction with their clients must be converted into API calls to modify Pounamu data structures. Figure 21 shows a sequence diagram outlining the process of converting user interactions into such Pounamu data structure updates. Again this is slightly different for each kind of thin client. The request to generate the Pounamu API messages is done differently depending on the kind of client device and whether or not client-side scripting is used. A POST from the browser (either script-generated or user-generated) is processed by appropriate Pounamu/Thin components: User POST operations (button click in browser) 2a); client-side script POST operations; and MUPE XML messages are all translated into appropriate Pounamu diagram update Command objects (3a-c) e.g. AddShape, MoveShape, DeleteConnector or composite lists of Commands. These Command objects are sent to the Pounamu Server via RMI API call(s) (4) and are run by the Pounamu modelling tool server (5) to effect diagram updates. Diagram objects (shapes, connectors and the diagram itself) are modified by each operation (6), and underlying shared model instances (entities and associations) are modified appropriately (7). Diagrams sharing modified model information are then updated by further Command objects generated by Pounamu (8-9). A subscribe-notify mechanism is used by the Pounamu/Thin server diagram XML cache to refresh the cache contents for all modified diagrams in the Pounamu server (10). New client data is then generated and returned to the client device using the mechanism described in Figure 19.

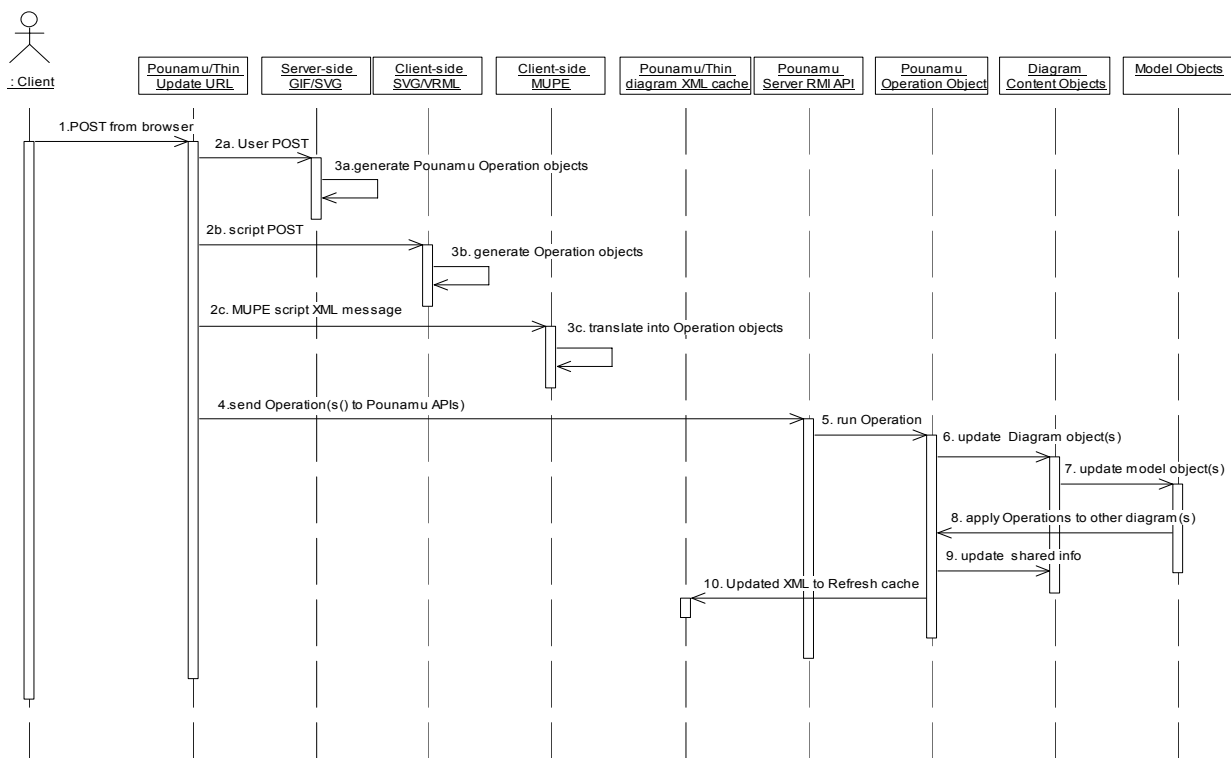


Figure 21. Update request processing from client device.

IMPLEMENTATION

Figure 22 provides an outline of the key technologies we have used to date to implement the Pounamu/Thin thin-client diagramming architecture. We chose to use a set of Java Servlets hosted by a Tomcat servlet engine and RMI communication between the servlets to a single instance of the Pounamu meta-tool acting as a shared application server. This Pounamu instance acts as both the meta-tool capability, allowing definition and modification of diagramming tools, and the data and processing application server for Pounamu/Thin.

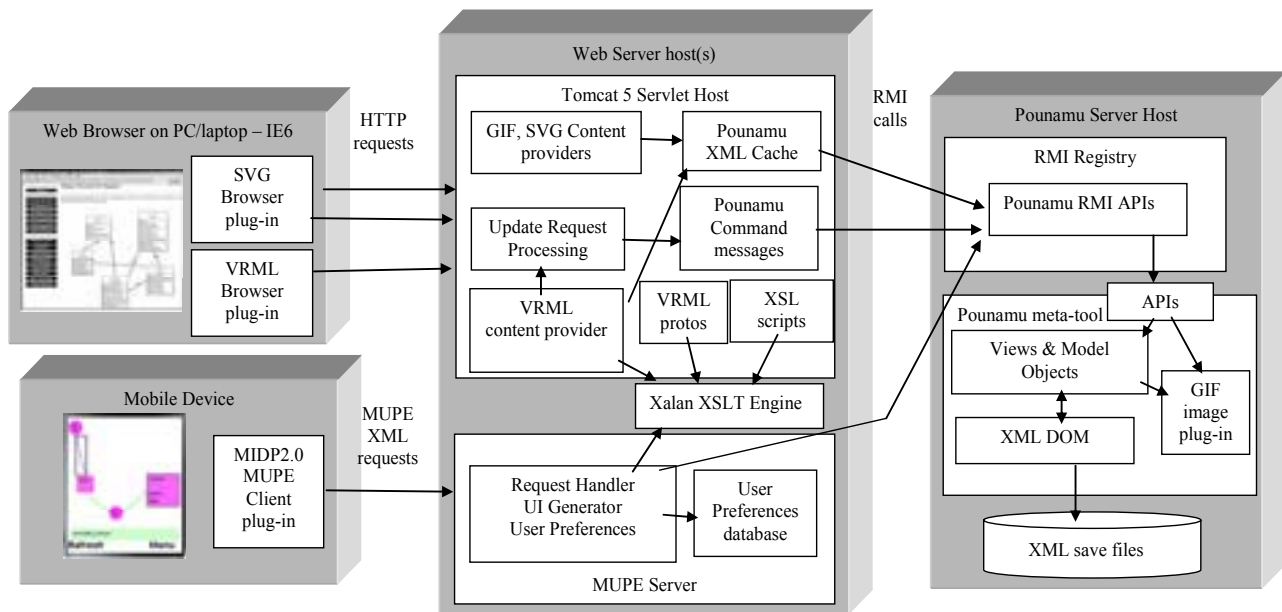


Figure 22. Pounamu/Thin APIs and technologies used.

Pounamu itself is implemented in Java and provides comprehensive RMI and web services interfaces (Zhu et al, 2004; Mehra et al, 2004). The RMI interface to Pounamu performs much faster than the web services one and we chose to use this for Pounamu/Thin. Java Swing was used to implement Pounamu's thick-client view editors and Pounamu makes extensive use of XML DOM structures to manage diagram and model data structure representations. We found it easiest to add a plug-in component to Pounamu to support generation of GIF images from its diagram views, as this was simple to do with its Swing components. This was used to provide the GIF images of views for the GIF-based rendering of diagram components. This facility is also currently used by the thick-client tool for both printing Pounamu diagrams and copying them to other applications, which turned out to be a useful side-benefit of this approach.

The SVG diagram renderer and editor is implemented quite differently to the GIF renderer. The latter simply requests that Pounamu return a GIF image of a view each time a view is changed. It then generates HTML for the view which includes editing command buttons, property sheet labels and text fields, and the GIF image. The SVG renderer instead requests an XML-encoded version of the view from the Pounamu server. It then traverses the XML and generates an SVG encoding of the Pounamu shape and connector objects in the view XML, along with HTML for view and property editing as per the GIF-based renderer. The SVG is rendered by the browser plug-in, resulting in more polished, detailed diagram appearance than the GIF-based version. We implemented the SVG translator in Java rather than using XSLT transformation scripts as we found using Java provided much faster performance and because this process required quite complex calculation algorithms not well-suited to implementation in XSLT. The multiple edit buffer facility uses the web component-stored SVG XML data, meaning buffered edits do not need to be sent to the Pounamu application server and are hidden from other users until committed. This wasn't possible to do with the GIF-based version as it has to use the Pounamu application server to generate diagram GIF images, whereas the SVG version generates SVG data in the web components.

As the structural transformations were relatively straightforward, we used XSLT to support the generation of VRML renderings of Pounamu diagrams. The VRML content provider servlet requests a Pounamu diagram's XML content and applies a set of XSLT transforms to the XML to generate a X3D (VRML XML) diagram. One VRML prototype shape must be defined for each Pounamu diagram shape and connector type which is used by the transforms to generate a VRML element for each Pounamu diagram element. Currently we do not provide a tool to generate these automatically from Pounamu meta-tool shape and connector specifications as we do for SVG and MUPE content. We found the range of possible renderings of 2D Pounamu components using 3D VRML elements to be too complex for fully automatic transformation as we did with SVG and MUPE diagram content. One straightforward future extension would be to add automatic generation of a first-attempt prototype VRML shape and allow users to modify these prototypes with a VRML authoring tool.

Pounamu views are saved to an XML format by the tool for both saving and loading views and also to support collaborative thick-client editing of views via heavyweight distributed message passing between multiple instances of Pounamu (Mehra et al, 2004). In addition, Pounamu view editing Command objects can be converted to and from an XML format, which was done in earlier work to support thick-client Pounamu tool collaborative view editing. We used

both of these facilities to support thin client editing of Pounamu views and the SVG-based view rendering. The Pounamu/Thin diagram editing facilities convert user interactions with diagrams and property forms into XML-format Pounamu Command objects. These are sent by the RMI API to the Pounamu tool acting as Pounamu/Thin application server and are run on the appropriate view. We synchronise access to the Pounamu application server in the servlet components to ensure only a single user's list of Command objects are exchanged with Pounamu at a time, to ensure consistent transactional behaviour.

We chose the Nokia MUPE mobile device technology to implement our Pounamu/Thin mobile thin-client diagramming architecture with Java RMI communication to Pounamu. The MUPE architecture provides a rich mobile client development platform that is device-independent. It also provides a server with a Java-based plug-in architecture, enabling a wide range of server application components to be developed. We developed several of these to implement Pounamu/Thin support for mobile devices. The Pounamu/Thin mobile device server is made up of a set of MUPE server components that interact with the Pounamu application meta-tool to synthesise mobile device user interfaces. MUPE client content is an XML document containing layout, content and scripting information.

The Pounamu/Thin mobile device server comprises 3 key components: Request Handling, UI Generation, and User Tool Configuration. Request handlers respond to all requests from MUPE clients. They hold the functional services supplied by the server. The Request Handling component is responsible for handling user requests, communicating with the Pounamu/Thin server components, and notifying corresponding UI generators to generate pages. The UI generators generate MUPE format user interfaces of Pounamu views, property forms, and tool configurations. Each UI generator contains a set of objects, which have their own visual representation templates in an XML format and are similar to VRML prototype objects. Each registered user has his/her own tool configuration as an XML file for each tool he/she has loaded. These configuration files are stored in the Pounamu/Thin mobile server. To generate a diagram view, the Model UI generator collects and combines information from multiple sources, including model view information from the Pounamu application server, user personalized specification of diagrams from user's tool configuration file, screen size of client device, and diagram rendering templates of rendering objects.

We initially implemented custom Java code to carry out this UI generation. However, like the VRML content generator we modified this to use XSLT transformations to make them easier to modify and extend, at the cost of some performance loss. Our aim is to replace the MUPE server with Tomcat servlets that provide MUPE-compatible XML to mobile devices and use XSLT-based transformations to process requests. This refactoring will complete the unification of the SVG, VRML and MUPE request/render component architecture.

Limited support is provided for collaborative diagram editing. A cache is used by the SVG and VRML content renderers to reduce requests on the shared Pounamu application server. The RMI plug-in for the Pounamu application server sequences all requests from the Pounamu/Thin web server components ensuring no concurrent modification of Pounamu data structures. A sequence of Commands can be run as a unit to ensure transactional integrity of a set of diagram modifications. Plug-ins in the application server control concurrent access to diagram components via locks and highlighting of diagram elements that are in use by another user. These were developed to support thick-client collaborative editing (Mehra et al, 2004) but also work well for thin-client collaborative editing.

Integration with other software tools is provided by Pounamu, typically using XML-based data exchanges. In addition, some users can choose to edit Pounamu views using the existing thick-client diagram editor. In this case, they run a version of Pounamu on their own workstation with a copy of all model and view information they share. The web services API is used to synchronise their views with the "shared" views used by the thin-client diagram editing web components. We don't discuss this approach further in this paper, but note that it is possible with our architecture to have a mix of users using either Pounamu/Thin thin-client or thick-client Pounamu diagramming of the same shared view.

EXPERIENCES

To date we have built several domain-specific language applications to leverage our Pounamu/Thin thin-client user interface synthesis technology. In conjunction with an industry client producing project management tools we developed several diagram specifications in Pounamu for use in web browsers and mobile devices (Zhou et al, 2006). These included Gantt charts, PERT charts and work break-down schedules. We have developed a thin-client software architecture design tool for use with the MASE project (Chau and Maurer, 2004) and to allow user configuration of Fitness tests. We developed a thin-client XForm design tool with an industry client to support design of forms in a web browser with generation of XForm descriptions from Pounamu design diagrams. We have used Pounamu/Thin with several in-house software design tools including a Pounamu UML tool with class, collaboration, use case, sequence diagram and deployment diagram views, and a business strategy modelling tool with process modelling views (Cao et

al, 2005). In no case has any code change needed to be made to Pounamu or Pounamu/Thin components to provide fully-functional web browser and mobile phone diagram editors for these tools.

We have carried out two evaluations of Pounamu/Thin thin-client web-based diagramming, in addition to reviewing and comparing our evaluation results to those of other thin-client diagramming tools. Our first evaluation was a user survey via a questionnaire of nine experienced UML users, all either senior academics, experienced industry software designers or post-graduate students. They were asked about their experiences using the thick-client Pounamu-implemented UML diagramming tool, the GIF-based thin-client UML tool, and the SVG-based thin-client tool with multiple edit buffering. A single UML tool was specified in Pounamu and the exact same specification used in all three tools for the survey. The UML tool provided relatively complete class diagrams, collaboration diagrams, sequence diagrams and deployment diagrams. A set of simple single-user and collaborating user UML modelling tasks was set for these users. The single user tasks included taking a simple software design, an on-line video rental system, and building structural and behavioural models for the system using each of the UML tools. Users were split into three groups of three, each group using the three UML tools (thick client, GIF and SVG) in different orders. The collaborative task we set for groups of three users was to review a design for the video system and to make simple enhancements to it.

Feedback from this survey indicated that the users perceived the thin client tool provided the same facilities as the thick-client version. Most users in the evaluation felt that they would be happy to use any of the three approaches, the thick-client using Java Swing, the GIF-based thin-client version or the SVG-based version. Feedback on user interaction in the thin-client tools was positive, with users able to create and modify diagram components easily. Novice Pounamu users suggested that the learning curve for the thin-client tools was less than the thick-client one and that it was an advantage that they needed no installation and configuration to use. The learning curve for inexperienced users of Pounamu tools was found to be least for users of the GIF-based thin-client tool. However, users preferred the rendering of diagrams in the SVG tool rather than the GIF one. The buffering facility of the SVG-based tool confused some users while others liked it, particularly when collaboratively editing diagrams with others. It may be the case that further experience with it is needed and it can be turned off if not wanted.

The Response time of the thin-client tools was found to be acceptable despite when using full diagram and page refresh after every editing operation. Client-side scripting of some operations e.g. highlighting diagram elements in the SVG plug-in, significantly improves some aspects of response-time and provides a more common direct manipulation model for editing. All users preferred the client-side scripting support for direct manipulation of SVG content, specifically for moving and resizing shapes and adding shape connectors. Users perceived that the thin-client tools do not provide as good awareness facilities as the thick-client Pounamu tool currently does (Mehra et al, 2004). Currently changing a tool specification can only be done with the thick-client tool designer, which was seen as a disadvantage by some users.

Our second evaluation was a cognitive dimensions (CD) evaluation of the three versions of our UML diagramming tool. The CD framework provides a way of assessing various characteristics of visual languages and tools along a variety of usability “dimensions” (Green, 1989). We compared the Pounamu/Thin tool characteristics to those of the original Pounamu thick-client tool to determine their key similarities and differences. We summarise the results of our CD evaluation below.

- *Viscosity*. The thin-client diagrams are generally more “viscous” i.e. harder to effect change, than their thick-client counterparts. This is particularly so for moving and resizing operations that require multiple web browser interactions by users without client-side scripting. This characteristic is cited most commonly against use of thin-client diagramming approaches. However, our evaluation of Pounamu/Thin with end users indicates supporting client-side drag-and-drop for resize, move and delete generally mitigates this problem for most users. Diagram editing on MUPE mobile devices similarly must leverage a degree of client-side scripting to support zoom and pan in addition to resize/move operations.
- *Visibility and Juxtaposability*. The Pounamu thick-client UML modeller allows multiple views to be displayed side-by-side or accessed via both a tree and tabbed panes. Thin-client tools require multiple web browser windows displaying different views to be opened and positioned on the user’s display and a simplified list of views is provided for the user to select from. The mobile thin client permits only one view to be visible at any time and thus provides poorer visibility and juxtaposability of diagrams than the other approaches due to its small screen size limitations.
- *Hard mental operations*. Users found the learning curve of our Pounamu/Thin tools to be somewhat less in many respects than the Pounamu thick-client tool interface. Pounamu/Thin interfaces present many interaction options in more explicit ways, via always-visible buttons, a large message panel and frame-based property editing, rather than pop-up context-sensitive menus in the thick-client version. However, some operations such as moving and resizing require non-intuitive sequences of operations in some Pounamu/Thin configurations. Navigating a generated VRML

3D model for a Pounamu diagram uses non-intuitive mouse and keyboard manipulations for the VRML browser plug-in. Interaction with 3D diagram elements is similarly non-intuitive at times due to the synthesized Z dimension for some shapes and the un-obvious interaction points on shapes.

- *Hidden dependencies and View support.* Multiple views of model elements are supported in all diagramming tools and dependencies between view elements and model elements are via shape and connector names. Pop-up menus on thick-client shapes allow access to linked information, including other views they appear in, but the thin-client tools currently require access via buttons and non-contextual option lists. Mobile hosted diagrams must incorporate multi-level zoom support in addition to the ability to move to other diagrams, resulting in loss of context for complex diagrams or frequent inter-diagram navigation.
- *Consistency.* The thin-client diagramming tools adopt as their main interaction metaphor page-based diagram viewing and interaction, where each user interaction causes a POST to the web server and then refresh of the screen. The Pounamu thick-client diagramming interface instead uses direct manipulation, drag-and-drop and pop-up menu non-modal approach. The thin-client's page-based approach thus supports less direct-manipulation activity but fits a user's mental model of web page post/display behaviour that is common to all web browser-hosted applications. The button-based, page-organised approach of Pounamu/Thin to providing editing and view management operations provides users with a less complex interface than the many thick-client Pounamu tool pop-up and pull-down menus. Once users are familiar with the request-response approach adopted by Pounamu/Thin interfaces for e.g. creating shapes, the exact same metaphor is used with all other editing and view selection operations.
- *Progressive Evaluation.* The SVG thin-client diagram tool's buffered input approach supports user-controlled editing transactions, allowing the user to indicate when a set of edits are complete and they want them actioned on the shared diagram vs their own copy of the diagram. The MUPE mobile and VRML thin-client diagramming applications in contrast post user updates to the server asynchronously via their client-side scripting to effect server diagram updates. These mechanisms become apparent when performing collaborative work where the thin-client tools require user-directed refresh of pages to show highlighting of others' updates and locked diagram elements. In contrast, the thick-client Pounamu UML tool uses push-based view update and redisplay, providing a more natural, proactive collaborative work supporting user interface (Mehra et al, 2004).

We have evaluated our Pounamu/Thin MUPE mobile device components using several visual modelling tools specified by Pounamu. Again no code change to Pounamu/Thin was required. All that was needed was a configuration specification by the end user for each view type if s/he wanted to modify the default configuration. We have carried out a preliminary evaluation of the usability and performance of these Pounamu/Thin mobile device-based tools with several experienced mobile phone and PDA users. We carried out the experiment with the Pounamu/Thin and Pounamu servers installed on a workstation and users accessing the mobile interface via a MUPE client simulator, also running on separate workstations in different locations. We used two basic experiments, one using the project management tool prototype with users carrying out basic task planning and co-ordination activities. The second involved users reviewing and making minor modifications to (fairly simple) UML class diagrams representing a data model for a business application. Users carried out both concurrent (synchronous) activities as well as asynchronous activities in both experiments. Users also used the Pounamu desktop application to carry out these same tasks as a comparison. We used the exact same tool specifications from Pounamu for both Pounamu/Thin experiments as for the desktop viewing and editing experiments.

These evaluations indicate that our Pounamu/Thin mobile device diagram approach provides similar viewing and editing capabilities to our previous desk-top thick client and web-based thin client versions of Pounamu. Users were able to access both navigation and editing facilities in the mobile device but as expected the editing of diagrams was much more difficult than desktop or even web browser-based interfaces. Navigation issues on a small screen were to some degree mitigated by Pounamu/Thin's multi-level zoom capability and by users being able to specify multiple visual representations for a single diagram shape. Users found these features to be essential in order to support complex diagram browsing and drill-down to detailed item viewing and editing. Users indicated that more automated support by the Pounamu tool would help diagram editing e.g. automated placement, layout and auto-zooming both during browsing and editing. This was particularly so for the UML tool which provides little such support, as it is not required by users in the desktop and web browser interfaces. The Gantt chart tool provides some automatic layout support which users found helpful on the mobile device. In both examples as soon as diagrams become moderately large (greater than about 15 items) they became very difficult to use on the mobile interface. Overall users thought the approach to be effective for making diagrammatic content accessible via a mobile device while allowing other users to simultaneously access desktop and web browser versions of the content. We are using the feedback obtained from this preliminary survey to refine our approach. It is clear from this feedback that a number of usability enhancements to make browsing and editing faster and easier are required to make the approach feasible for real work. These include user configurable hot-keys to speed up some functions, especially browsing, and user configurable layout in addition to zoom shape

specification. In addition, support for collaborative interactions needs to be improved, as has been done in the work of others (Luz and Masoodian, 2003). Nevertheless, the initial feedback is promising and justifies the development of the experimental testbed. Our aim is to conduct further experiments using this framework to assist us in developing guidelines for the types of diagrams able to be usefully deployed in this way and additional mitigation approaches to overcome the limited screen and interaction capabilities.

Like other researchers we have encountered some difficult technical challenges in developing Pounamu/Thin mobile device user interfaces. The graphic rendering limitations of MIDP and difficulty of user interaction on mobile phones means that diagrams specified and rendered on mobile devices cannot be as complex as on a PC. However, as mobile devices continue to become more powerful we expect to be able to achieve more complex diagramming rendering on mobile devices. Icon specification is currently quite cumbersome using the MUPE mobile device interface. This would be much easier and flexible on pen-based PDAs than on the key-based phones that we have used in our experiments to date. Because of the memory limitations on current mobile devices, very large diagrams cannot be rendered, even when using MUPE thin client technology. Instead, at present, smaller overlapping Pounamu views must be constructed. Again the rapid evolution of mobile hardware will remove this limitation. We currently use a basic layout algorithm to transform Pounamu views into an overview for mobile devices. This works fine for e.g. our UML tool which is not so sensitive to spatial layout. However, some tools use layout as an important component of the diagram such as the Gantt charts in our project management tool.

Preliminary experiments have been carried out with loading the Pounamu/Thin web components. We simulated multiple user interaction with the diagramming tools by setting up the Pounamu application on one host and Tomcat server hosting Pounamu/Thin servlets on another machine with a 100MBit network connection. Several concurrent client diagram display and update operations were simulated by a multi-threaded Java application invoking Pounamu/Thin components with diagram request and update operations. We used the server-side SVG diagramming servlets as the target Pounamu/Thin components, assuming no client-side editing support. Performance results indicate a response time to the user of between 0.5-1.5 seconds can be supported for only around 4-7 concurrent users depending on the complexity of the editing operations being performed and the size of the diagram. The current architecture of Pounamu/Thin thus provides good response time for a small number of concurrent users, typical of what is needed for shared design diagram editing. The web components, both in the Tomcat and MUPE servers, can be multi-threaded and the servers themselves can be replicated to provide load-balancing and fault-tolerance. Caching of Pounamu XML diagram data structures in the web server also reduces requests for this data from Pounamu and allows a large number of users to concurrently browse, but not edit, diagrams. Using client-side scripting minimises server loading for simple operations such as move and resize of shapes, but adding shapes and connectors and modifying their properties must produce web component POST operations, which then result in Pounamu application server RMI calls. SVG, VRML and MUPE all support client-side DOM representations of their screens. These diagram data models could be incrementally updated instead of completely re-displayed after editing operations. For large diagrams this would significantly reduce the time to re-render the diagram in the client but also reduce network traffic between the client and Pounamu/Thin server as only changed document content could be sent to the server.

As all diagram update requests to the Pounamu application server must be serialised, there is limited scope for scaling this part of the system for editing any particular model. The Pounamu meta-tool holds both the tool definitions and shared diagram and model data structures and applies complex logic when updating diagrams to modify the model and related diagram elements. This means while editing requests can be buffered in the client browser by scripts or in the web server for buffered editing, eventually they must be applied to the shared application server via its RMI API. Any buffering of edits runs the risk of concurrent user modification of the same diagram elements and either lost updates or the need to merge multiple updates. In earlier work we used a peer-to-peer system to provide collaborative editing for multiple users of the thick-client Pounamu tool (Mehra et al, 2004). It may be possible to use a similar approach to have multiple Pounamu applications running and load-balance Pounamu/Thin requests across these, using the peer-to-peer collaborative editing support infrastructure to keep the Pounamu application data consistent in an asynchronous fashion. Replication of the Pounamu server is also possible when editing different models provided tool specifications are not being concurrently altered by one of the replicated Pounamu instances. This is the most viable short term route to scalability, at the expense of liveness of tool specifications, as only limited numbers of users are likely to be editing any one model at a time.

Key areas for future work we are planning include extending our work to cover more types of devices with other styles of input, such as stylus input on PDA and Tablet PC and speech input. We are keen to see how diagram-based visual design tools benefit from these alternative interaction mechanisms. As we discussed above, specifying representations of diagrams will definitely be improved by using stylus input.

A key issue we have had to address for each of our Pounamu/Thin components is *how much computation is deployed to the client*, or in other words, *how thin can the client be while still being usable?* In the web browser case, we

experimented with a “fully thin” approach and one combining additional plug-ins (SVG and VRML) and scripting. It is clear from our user evaluations that the fully thin case leads to low usability rating by end users due to its inherent latency problems. The issue then is what technology provides sufficient local computational power for usability, while minimising both payload cost, and versioning issues. Our feeling is that there is no correct answer here, and that a variety of technologies could and should be used for this purpose. Our SVG + ECMA script approach would be quite similar in approach to one based on Ajax/DHTML, Laszlo/Flash, or small applet/DOM. Similarly for the mobile deployment case, the MUPE MIDP client provided sufficient computational power for our needs but other clients could be equally effective. This suggests a program of future work opportunities around refactoring and generalising the Pounamu/Thin platform.

Unifying the Pounamu/Thin content generation and request handling components into a single set of Tomcat web server components is desired to enable sharing of request processing, content generation and support for individual user preferences. At present, the GIF and SVG generators use custom code to generate content whereas XSLT or another transformation/content generation technologies like Eclipse Java Emitter Templates or Apache Velocity could be alternatives. Similarly, only the MUPE-based mobile clients allow end users to specify their own custom rendering for shapes and this would be useful in SVG and VRML-based renderers. In addition, we believe the multi-level zooming facility employing multiple levels of detail used by the MUPE-based mobile device interfaces would be useful in Pounamu tools in general. We are investigating generating Laszlo interface descriptions instead of SVG as these compile to Macromedia Flash implementations providing much richer web browser interactions and content, while minimising some of the browser inconsistency issues we have noted. We would like to apply our approaches to mobile technologies other than MUPE, such as Mobile SVB, Symbian development framework in C++, and Microsoft .Net Mobile version. We want to improve the layout algorithm used for overview diagrams to better produce layouts for those diagram types that rely heavily on spatial relationships. Potential approaches may include spatial reasoning and other related approaches on mobile devices (Wobbrock et al, 2004). We would like to integrate some of our previous work on collaborative editing support (Mehra et al, 2004) to all the Pounamu/Thin user interfaces. This would provide richer support for group awareness and synchronisation between multiple users of a Pounamu tool. Further investigation is needed to enable scaling of the system architecture beyond a handful of concurrent users. Finally, we see our work as one step in the evolution of a more generic framework for multi-device, multi-platform diagram editing and visualisation support. The work reported here has provided the basis for such a general component based framework. However, using the framework as it currently stands is cumbersome, requiring significant programming effort to introduce a new interface technology, such as Ajax, Laszlo or Symbian. Our next major step, therefore, is to refactor the framework and abstract a set of configuration languages and plug-in points permitting more straightforward adaptation of the framework both for the addition of new thin client interface technologies and for use by metatools other than Pounamu.

SUMMARY

We have described our experiences in developing architectures and solutions for thin client modelling and diagramming tools. Our approach has been to develop several exemplar thin client interfaces to a metatool permitting any tool specified and generated by the metatool to be deployed, without additional programming effort, using any of those thin client interfaces. We have developed interfaces for GIF, SVG and VRML, using web browser clients, and for MIDP compatible phones, using Nokia’s MUPE technology. In the process we have experimented with combinations of client and server side scripting and undertaken a number of user evaluations of the effectiveness of our approaches, all demonstrating that effective thin client interfaces can be automatically generated for tools specified using the metatool. From this experience we have partially refactored our exemplar solutions into a more generic framework, with significant component reuse, and are now working on a more complete refactoring that will lower the cost of reusing the framework through the addition of a set of configuration languages and plug-in points.

ACKNOWLEDGEMENTS

The authors acknowledge the assistance of the New Zealand Foundation for Research Science and Technology in funding this project. We also acknowledge the support of our industry partners in providing relevant tool case studies, and our evaluation subjects who willingly gave their time. Hermann Stoeckle acknowledges the University of Auckland Computer Science Department for scholarship assistance.

REFERENCES

1. Baudischl, P., Xie, X., Wang C., and Ma, W.Y. (2004): Collapse-to-Zoom: Viewing Web Pages on Small Screen Devices by Interactively Removing Irrelevant Content. In Proceedings of the 2004 ACM Conference on User Interface Software Technology.

2. Bentley, R., Horstmann, T., Sikkil, K., and Trevor, J. (1995): Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. *Proc. of the 4th International WWW Conference*, Boston, MA, December 1995.
3. Bonifati, A., Ceri, S., Fraternali, P., Maurino, A. (2000) Building multi-device, content-centric applications using WebML and the W313 Tool Suite, *Proc. Conceptual Modelling for E-Business and the Web*, LNCS 1921, pp. 64-75
4. Burnett, M. Atwood, J., Walpole, R. and Reichwein, J. (2001): Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, *Journal of Functional Programming*, 11(2), March 2001, pp. 155-206.
5. Buyukkoten, O., Garcia-Molina, H., Paepcke, A., and Winograd, T. (2000): Power browser: efficient web browsing for PDAs. In *Proceedings of CHI 2000 Conference on Human Factors in Computing Systems*, pp. 430-437.
6. Cao, S., Grundy, J.C., Stoeckle, H., Hosking, J.G., Tempero, E., Zhu, N. (2005): Generating Web-based User Interfaces for Diagramming Tools, In *Proceedings of the 2005 Australasian User Interfaces Conference*, Jan 31-Feb 3, 2005, Newcastle, Australia, *Conferences in Research and Practice in Information Technology*, Vol. 40.
7. Chalk P. (2000): Webworlds-Web-based modeling environments for learning software engineering, *Computer Science Education*, vol.10, no.1, 2000, pp.39-56.
8. Chau, T. and Maurer, F. (2004): Tool Support for Inter-Team Learning in Agile Software Organizations, in *Proceedings of the Workshop on Learning Software Organizations 2004*, Springer.
9. Chen, Y., Ma, W.Y., and Zhang, H.J. (2003): Detecting Webpage Structure for Adaptive Viewing on Small Form Factor Devices. In *Proceedings of WWW 2003*, pp 225-233.
10. Cox, P.T., Giles, F.R. and Pietrzykowski, T. (1989): Prograph: a step towards liberating programming from textual conditioning, In *Proceedings of the 1989 IEEE Workshop on Visual Languages*.
11. Ebert, J., Siittenbach, R., Uhe, I. (1997): Meta-CASE in practice: A case for KOGGE, *Proc. 9th International Conference on Advanced Information Systems Engineering*, LNCS 1250, Barcelona, Spain, Springer-Verlag (1997) 203-216.
12. Eisenstein, J. and Puerta, A. (2000): Adaptation in automated user-interface design, *Proc. 2000 Conference on Intelligent User Interfaces*, New Orleans, 9-12 January 2000, ACM Press, pp. 74-81.
13. Evans, E. and Rogers, D. (1997): Using Java Applets and CORBA for multi-user distributed applications, *Internet Computing* 1(3), 1997, IEEE CS Press, pp. 43-55.
14. Ferguson, R.I., Parrington, N.F., Dunne, P. Hardy, C., Archibald, J.M. and Thompson, J.B. (2000): MetaMOOSE - an Object-Oriented Framework for the construction of CASE tools, *Information and Software Technology*, vol. 42, no. 2, January 2000, pp. 115-128.
15. Gordon, D., Biddle, R., Noble, J. and Tempero, E. (2003): A technology for lightweight web-based visual applications, *Proc. of the 2003 IEEE Conference on Human-Centric Computing*, Auckland, New Zealand, 28-31 October 2003, IEEE CS Press.
16. Graham T.C.N., Stewart, H.D., Kopae, A.R., Ryman, A.G., Rasouli, R. (1999): A World-Wide-Web architecture for collaborative software design, *Proc. of the Ninth International Workshop on Software Technology and Engineering Practice*, IEEE CS Press, 1999, pp.22-29.
17. Green, T.R. (1989) Cognitive dimensions of notations, *People and Computers V*, Sutcliffe, A. and Macaulay, L. Eds, Cambridge University Press, 1989.
18. Grundy, J.C., Mugridge, W.B. and Hosking, J.G. (2000): Constructing component-based software engineering environments: issues and experiences. *Information and Software Technology* 42, 2, January 2000, pp. 117-128.
19. Grundy, J.C. and Zhou, W. (2003): Building multi-device, adaptive thin-client web user interfaces with Extended Java Server Pages, In *Cross-platform and Multi-device User Interfaces*, Wiley, 2003
20. Iivari, J. (1996): Why are CASE tools not used?, *CACM*, vol. 39, no. 10, 1996, pp. 94-103.
21. Kaiser, G.E. Dossick, S.E., Jiang, W., Yang, J.J., Ye, S.X. (1998): WWW-Based Collaboration Environments with Distributed Tool Services, *World Wide Web*, vol. 1, no. 1, 1998, pp. 3-25.
22. Kelly, S., Lyytinen, K., and Rossi, M. (1996): Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, *Proc. of CAiSE'96*, Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.
23. Khaled, R., McKay, D., Biddle, R. Noble, J. and Tempero, E. (2002): A lightweight web-based case tool for sequence diagrams, *Proc. of SIGCHI-NZ Symposium On Computer-Human Interaction*, Hamilton, New Zealand, ACM Press, 2002.
24. Luz, S., and Masoodian, M. (2004): A Mobile System for Supporting Non-Linear Access to Time-Based Data. *Conference Proceedings of AVI 2004, The 7th International Working Conference on Advanced Visual Interfaces* (Gallipoli, Italy, 25-28 May), ACM Press, 454-457
25. Luz, S., Masoodian, M., and Weng, G. (2003): Browsing and Visualisation of Recorded Collaborative Meetings. *Conference Proceedings of HCI International '03, 10th International Conference on Human-Computer Interaction* (Crete, Greece, 22-27 June), Lawrence Erlbaum Associates, Vol. 2, 148-152.

26. Lyu, M. and Schoenwaelder, J. (1998): Web-CASRE: A Web-Based Tool for Software Reliability Measurement, *Proc. of International Symposium on Software Reliability Engineering*, Paderborn, Germany, Nov, 1998, IEEE CS Press.
27. Mackay, D., Biddle, R. and Noble, J. (2003): A lightweight web based case tool for UML class diagrams, *Proc. of the 4th Australasian User Interface Conference*, Adelaide, South Australia, 2003, Conferences in Research and Practice in Information Technology, Vol 18, Australian Computer Society.
28. Marsic, I. (2001): An architecture for heterogeneous groupware applications, *Proc. International Conference on Software Engineering*, May 2001, IEEE CS Press, pp. 475-484
29. Masoodian, M., and Budd, D. (2004): Visualization of Travel Itinerary Information on PDAs. *Conference Proceedings of AUIC 2004, The 5th Australasian User Interface Conference*, (Dunedin, New Zealand, 18-22 January), Australian Computer Society Inc, 65-71.
30. Maurer, F. and Holz, H. (2002): Integrating Process Support and Knowledge Management for Virtual Software Development Teams, *Annals of Software Engineering*, vol. 14, no. 1-4, 2002, pp. 145-168.
31. Maurer, F., Dellen, B., Bendeck, F., Goldmann, S., Holz, H., Kötting, B., Schaaf, M. (2000): Merging project planning and web-enabled dynamic workflow for software development, *IEEE Internet Computing*, Volume 4 , Issue 3, May 2000, pp. 65-74.
32. McIntyre, D.W. (1995): Design and implementation with Vampire, *Visual Object-Oriented Programming*. Manning Publications, Greenwich, CT, USA, 1995, Ch 7, 129-160.
33. McWhirter, J.D. and Nutt, G.J. (1994): Escalante: An Environment for the Rapid Construction of Visual Language Applications, *Proc. VL '94*, pp. 15-22, Oct. 1994
34. Mehra, A., Grundy, J.C. and Hosking, J.G. (2004): Adding Group Awareness to Design Tools using a Plug-in, *Web Service-based Approach, Proc. of the 6th Int. Workshop on Collaborative Editing Systems*, Chicago, 5th Nov 2004.
35. Minas, M., and Viehstaedt, G. (1995): DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, In *Proceedings of VL '95*, 203-210 Sept. 1995.
36. MUPE (2006) : Multi-User Publishing Environment (MUPE), www.mupe.net, June 2006.
37. Palm Corp. (2001): Web Clipping services, <http://www.palmos.com/dev/tech/webclipping/>, accessed 10th December 2006.
38. Quatrani, T. and Booch, G. (2000): *Visual Modelling with Rational Rose™ 2000 and UML*, Addison-Wesley.
39. Robbins, J., Hilbert, D.M. and Redmiles, D.F. (1998): Extending design environments to software architecture design, *Automated Software Engineering* 5 (July 1998), 261-390.
40. Rossel M. (1999): Adaptive support: the Intelligent Tour Guide. 1999 Int. Conf. Intelligent User Interfaces. ACM. 1999, New York, NY, USA.
41. Stephanidis, C. (2001): Concept of Unified User Interfaces, In *User Interfaces for All - Concepts, Methods and Tools*, Laurence Erlbaum Associates, pp. 371-388.
42. Sun, J., Dong, J.S., Liu, J. and Wang, H. (2001): An XML/XSL Approach to Visualize and Animate TCOZ. *Proc. of the 8th Asia-Pacific Software Engineering Conference*, Macau SAR, China, December 2001, IEEE Press, pp. 453-460.
43. Sun Microsystems, Project Looking Glass, http://www.sun.com/software/looking_glass, accessed October 20 2006.
44. Van der Donckt, J., Limbourg, Q., Florins, M., Oger, F., and Macq, B. (2001): Synchronised, model-based design of multiple user interfaces, *Proc. 2001 Workshop on Multiple User Interfaces over the Internet*.
45. Wobbrock, J., Forlizzi, J., Hudson, S., Myers, B. (2002): WebThumb: interaction techniques for small-screen browsers. In *Proc. UIST '02*, pp. 205–208.
46. Zhao, D., Grundy, J.C. and Hosking, J.G. (2006): Generating mobile device user interfaces for diagram-based modelling tools, In *Proceedings of the 2006 Australasian User Interface Conference*, Hobart, Australia, January 2006
47. Zhu, N., Grundy, J.C. and Hosking, J.G. (2004): Pounamu: a meta-tool for multi-view visual language environment construction, *Proc. of the 2004 International Conference on Visual Languages and Human-Centric Computing*, Rome, Italy, 25-29 September 2004, IEEE CS Press.

6.2 Engineering plug-in software components to support collaborative work

Grundy, J.C. and Hosking, J.G. Engineering plug-in software components to support collaborative work, *Software - Practice and Experience*, Vol. 32, No. 10, August 2002, Wiley, 983-1013

DOI: [10.1002/spe.472](https://doi.org/10.1002/spe.472)

Abstract: Many software applications require co-operative work support, including collaborative editing, group awareness, versioning, messaging and automated notification and co-ordination agents. Most approaches hard-code such facilities into applications, with fixed functionality and limited ability to reuse groupware implementations. We describe our recent work in seamlessly adding such capabilities to component-based applications via a set of collaborative work-supporting plug-in software components. We describe a variety of applications of this technique, along with descriptions of the novel architecture, user interface adaptation and implementation techniques for the collaborative work-supporting components that we have developed. We report on our experiences to date with this method of supporting collaborative work enhancement of component-based systems, and discuss the advantages of our approach over conventional techniques.

My contribution: Developed initial ideas for this research, co-designed approach, wrote and evaluated the software, wrote majority of paper, co-lead investigator for funding for this project from FRST

Engineering plug-in software components to support collaborative work

JOHN GRUNDY

*Department of Electrical and Electronic Engineering and Department of Computer Science,
University of Auckland, Private Bag 92019, Auckland, New Zealand
john-g@cs.auckland.ac.nz*

JOHN HOSKING

*Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
john@cs.auckland.ac.nz*

ABSTRACT

Many software applications require co-operative work support, including collaborative editing, group awareness, versioning, messaging and automated notification and co-ordination agents. Most approaches hard-code such facilities into applications, with fixed functionality and limited ability to reuse groupware implementations. We describe our recent work in seamlessly adding such capabilities to component-based applications via a set of collaborative work-supporting plug-in software components. We describe a variety of applications of this technique, along with descriptions of the novel architecture, user interface adaptation and implementation techniques for the collaborative work-supporting components that we have developed. We report on our experiences to date with this method of supporting collaborative work enhancement of component-based systems, and discuss the advantages of our approach over conventional techniques.

KEYWORDS: software components, groupware, collaborative work tools, software architecture, user interface adaptation, aspect-oriented design

INTRODUCTION

Many software applications, such as CASE tools, programming environments, process and project management tools, and distributed Information Systems, require collaborative work facilities. Such facilities include support for both synchronous and asynchronous editing of documents; group awareness facilities; annotation and versioning of documents; messaging, email and chat; change notification and other task automation facilities, and process-based work co-ordination. Because of this need for collaborative work support, many frameworks, toolkits and application generators have been developed to help construct tools with collaborative work, or “groupware”, facilities. Examples include the groupware toolkits GroupKit¹, Clockworks^{2, 3}, Rendezvous^{4, 5}, Suite⁶, and COAST⁷; architectures like Clock³, MetaMOOSE⁸ and ALV⁵; and extensible groupware tools like Teamwave⁹, CocoDoc¹⁰, ConversationBuilder¹¹, and MS Netmeeting^{TM12}. Many groupware-enabled tools have also been developed using ad-hoc techniques, such as Grove¹³, Mjolner¹⁴, Oz¹⁶ and SPADE¹⁵.

While these approaches allow developers to build systems that support various kinds of collaborative work, they each have key disadvantages. Groupware toolkits typically support only limited forms of groupware facilities, such as real-time conferencing (e.g. GroupKit), or groupware must be built into custom user interfaces and architectures, resulting in portability, extensibility and performance problems (e.g. Rendezvous, Suite). Most “extensible” groupware systems restrict new tools to those built with the system’s architecture (e.g. ConversationBuilder, Teamwave, COAST). A major problem with most systems is that groupware facilities are built-in and not dynamically deployable and extensible (e.g. Netmeeting and CocoDoc), resulting in unnecessary architectural overheads when some groupware facilities are not needed by end users, and applications with “fixed”, non-expandable groupware functionality.

To solve these problems we have been developing plug-in software components to support the addition of groupware facilities to applications, even while they are in use. We have developed many such components and reused them in a number of component-based applications. Our approach is elegant, in that it does not require any modification to the code of the plug-in groupware components, nor the components of the application they are plugged into. Our approach differs from other component-based groupware systems, like COAST, CocoaDoc, Teamwave and Clock, in the variety of collaborative support possible, the openness of the architecture, the ability to add and remove groupware components dynamically, and the range of components and enhanced applications that we have developed.

In the following section we give some examples of component-based applications that require various plug-in groupware capabilities, and overview the requirements of such capabilities from application end user and application developer perspectives. We review related work strengths and weaknesses with respect to these groupware requirements. We then give examples component-based groupware user interfaces using some simple group work scenarios for different application domains, to demonstrate the versatility of application of our work. We then give an overview of the key kinds of groupware components we have identified and their basic architecture. We describe our own component-based framework we have used to build a number of groupware components and illustrate some groupware component architectures realised with this framework. We briefly discuss some of our groupware component designs and the implementation approaches we have used. We conclude by discussing the range of groupware components we have been able to build with this approach, results of reliability, performance and usability evaluations of some of these components, and areas for future work. We hope our experiences will be useful for others interested in developing their own component-based groupware applications.

MOTIVATION

Figure 1 shows some component-based software engineering tools and distributed information systems we have developed^{17, 18}. View (a) shows a workflow tool process diagram, used to describe workflow for software developers (but can be used to describe office automation workflow etc). View (b) shows an E-commerce application, a collaborative travel itinerary planner, we have developed¹⁸. This allows travel agents and customers to collaboratively plan a travel itinerary using a variety of tools, including an itinerary tree editor, itinerary item dialogues, itinerary visualisations and web browser. View (c) shows a CASE tool diagram under construction, used by software developers to aid the design of complex software systems. All of these tools were built by composing collections of “software components” i.e. reusable building blocks. Many components are shared between these quite different application domains (e.g. editing tools, data management and distributed system support, icons, event histories, and so on)¹⁹.

When using such applications, users require a wide range of “groupware” (collaborative work) facilities, to help them work effectively together^{6, 13, 11, 17}. While the travel itinerary planner is a very

different application to the software development tools, users require very similar kinds of collaborative work support. Such groupware functionality might include²⁰:

- *Collaborative work support.* This includes collaborative view editing, both synchronous and asynchronous (supporting multiple users editing CASE diagrams, workflow diagrams and travel itineraries together on-line or off-line). Versioning of edited information is necessary to support off-line work. “Group awareness” needs to be supported i.e. so other users can tell what a user is doing^{21, 13, 14, 1}.
- *Communication support.* This allows users to communicate about their work, for example text messaging, email, text and audio chat, video conferencing and note annotations on work artefacts^{22, 17}.
- *Co-ordination support.* Users need appropriate locking of view components currently being edited, highlighting of currently edited or recently edited components to avoid conflicts in their updates. Histories of work done need to be supported. Automatic notification of updates to views via communication mechanisms should be provided so users are told when others do “interesting things”. A shared work schedule (“to-do list”) is often desirable, along with work co-ordination via workflow tools^{21, 1}.

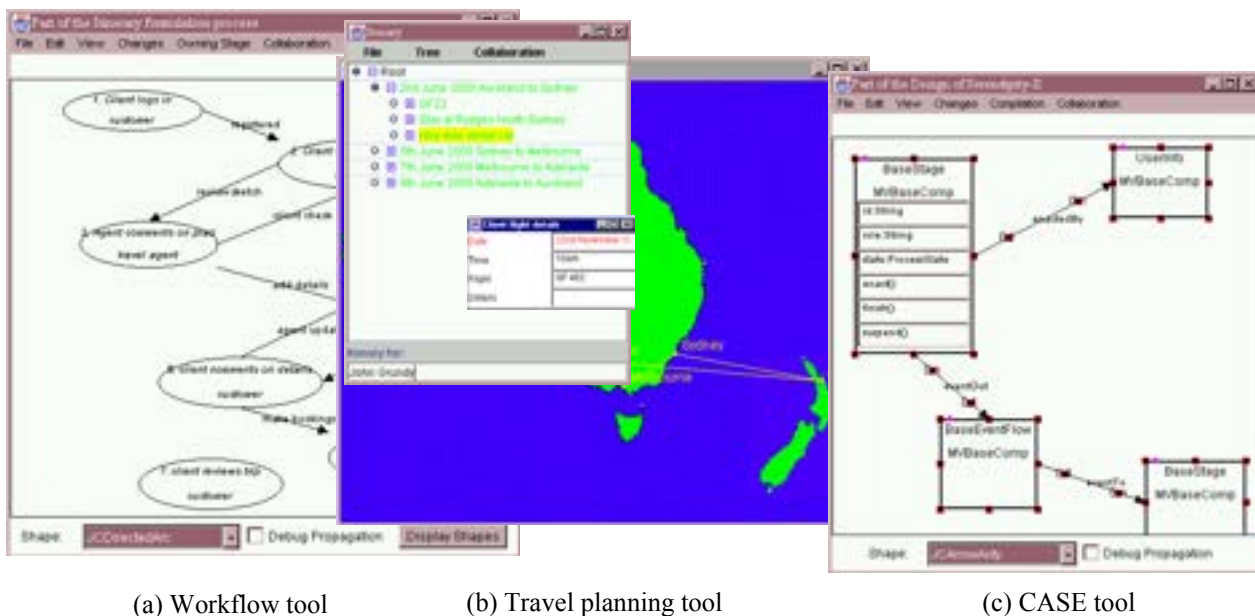


Figure 1. Examples of applications requiring groupware facilities.

When developing the systems illustrated in Figure 1, we have reused many software components, as reported elsewhere¹⁹. However, much of the collaborative work support these environments provide we originally built into the various framework abstractions and components we reused^{17, 19}. Thus while we used reusable components to build applications, our groupware support followed the “monolithic” application development approach, where developers had little control on what groupware support was included in specialised components, and this resulted in sometimes inappropriately reused and over-complex component functionality^{19, 18}. In addition, users were not given any control over what groupware functionality their environments supported, even if this was inappropriate to their specific needs.

The usefulness of this component-based approach to building tools such as those illustrated in Figure 1 leads to the question of whether it might be possible to develop components that embody the various collaboration facilities each of these applications (and others) require. Different kinds of collaborative work support could then be incorporated into discrete software components and be

able to be selectively reused, and even sometimes dynamically added to or removed from running applications as users require.

Such “groupware components” need to satisfy a number of key requirements to be effectively used in diverse systems. Groupware components need to:

- *Be deployable statically or dynamically.* Developers should be able to incorporate groupware components in applications at design and implementation time, and end users plug them into their running systems when in use.
- *Be seamlessly integrated with other software components.* Groupware components must utilise the existing event and method invocation interfaces of application components to provide the range of collaborative work facilities outlined above. Their user interfaces must also seamlessly integrate with the user interfaces of existing application components, even when dynamically deployed. This implies a need for components to provide appropriate methods to carry out such adaptations in a consistent, de-coupled manner.
- *Have discrete functionality.* Each groupware component should offer a distinct, preferably non-overlapping set of tailorable groupware facilities, allowing developers and end users to choose and configure a set of groupware components to provide the range of collaborative work facilities they require. Thus all groupware components should inter-operate in a seamless manner, and must also adapt each other’s user interfaces in a consistent manner.
- *Reuse suitable user interface, middleware and persistency components.* Where possible, groupware components should use similar or preferably the same packaged components that application components use to realise their user interface, communication (middleware) and data persistency needs. This reduces incompatibilities and redundancies in resulting groupware-enabled applications.

Many groupware applications have been built using ad-hoc approaches i.e. no specialised tool-kit or components. Examples include IRC and ICQ, Email, Grove¹³, Netscape’s Cooltalk, BSCW²³, Oz¹⁶ and SPADE¹⁵. The main advantage with using standard distributed systems programming APIs for groupware applications is flexibility. The main disadvantages include a lack of high-level, reusable abstractions and components for building such tools (making their construction very time-consuming and difficult), and an inability for users (and often developers) to extend or reconfigure the groupware functionality of these applications^{15,17}.

Because building groupware applications with standard APIs and frameworks is difficult, many groupware toolkits have been developed. Examples include GroupKit¹, Suite⁶, Rendezvous⁴, Meta-MOOSE⁸, and PCTE²⁴. These give developers built-in abstractions for constructing common groupware application facilities. Many successful groupware applications have been developed using these toolkits. However, most groupware toolkits suffer from “hard wired” facilities, a lack of extensible groupware facilities, and lack of ability of users to configure these facilities at run-time. In addition, applications must be built from scratch to use these toolkit facilities, and it is extremely difficult to integrate most groupware toolkit-built applications with existing applications, or extend existing applications with these toolkits. These problems lead to groupware applications with “fixed” functionality and often an inability of developers to provide some kinds of groupware facilities, as the toolkit they are using doesn’t support it. Conversely, many applications built using such toolkits often exhibit “groupware bloat”, with many of the groupware facilities provided by the toolkit unused but included in the applications anyway.

Component-based systems development technologies have become a popular approach to solving some common software development problems, such as lack of reuse and run-time configuration, and difficult-to-maintain, bloated, monolithic applications^{25, 26}. Examples of component development methods and technologies include Catalasys^{TM27}, Select Perspective^{TM28}, COM²⁹, Enterprise JavaBeans³⁸ and OpenDoc³⁰. Building groupware applications solely with standard

components shares the lack of abstractions and reusable components problem with using standard APIs and frameworks. However, some groupware applications have been developed using components alone, including CocoDoc¹⁰, TeamWave⁹, Netmeeting¹², and Orbit³¹.

The use of specialised component frameworks for building groupware applications has been shown to offer benefits. Examples of such frameworks include Clock³, our original JViews¹⁹, COAST⁷, Xanth³², and using CORBA and OODBMSs³³. Most of these approaches provide a component-based architecture with groupware functionality built into components in the architecture. Building groupware applications using such components is effective^{19, 7, 3}, but this still often results in problems such as fixed groupware functionality, component-bloat and lack of dynamically configurable and deployable groupware facilities. This is usually due to these groupware framework components themselves being hard-coded with specific kinds of groupware functionality. Most of these systems are oriented to quite limited problem domains e.g. synchronous groupware in COAST, tool integration in Xanth, and MVC-based synchronous editing environments in Clock. Most do not support user interface adaptations for components, making component integration difficult and resulting in poor quality interfaces. Some component architectures do support adaptive plug and play of components^{34, 35}, but to our knowledge have not been applied to component-based groupware development. Some aspect-oriented systems provide for the kinds of adaptations to running systems³⁶, but again to our knowledge have not been used for component-based groupware development.

OVERVIEW OF OUR COMPONENT-BASED GROUPWARE

In this section we illustrate various groupware component facilities using some simple scenarios based on extending the applications from Figure 1 with groupware capabilities. This was achieved by plugging groupware components into these three applications at run-time (and can also be removed at run-time).

Collaborative Editing

Consider John and Mark using a workflow system like that of Figure 1 (a) to describe their work processes (for some work domain – we use a simple software development process in the example below). They need to collaborate to view and edit these workflow diagrams. Initially they need to decide on the collaboration “level” – will they edit a workflow diagram synchronously (as one changes things the other immediately sees the changes made to their copy of the diagram) or asynchronously (they each edit independently then merge changes)? Figure 2 (1) shows John specifying, using a configuration menu of a “collaborating editing component”, “action”-level editing (this is semi-synchronous – changes are sent to Mark as John makes them and vice-versa but the other users(s) can make changes at the same time i.e. no locking/waiting for changes to be made to view). A message is sent to both John and Mark using a text message component (2), indicating the change in editing state for the shared diagram.

If John and Mark had previously been editing the view asynchronously, John may want to see changes Mark made off-line and merge them in with his copy of the view. This is done using an editing event history component (3). John can select edits to have applied automatically to his view, or may message Mark to discuss edits he disagrees with. Editing histories can be checked in/out (as can whole diagram copies) from a version control component (4). This allows workers to manage multiple versions of the same diagram/document using deltas (edit event groups) or copies of the entire work artefact (e.g. diagram). John may create a new version of the workflow diagram before doing further edits/applying Mark’s asynchronous edits to ensure the old information is not lost and can be later reviewed.

While editing the view, John and Mark need some cues as to what each is doing (so their edits don't clash). A multiple cursor component (5) shows where another user's cursor is and as Mark moves his mouse, this is refreshed on John's screen (typically each second or so). When Mark begins to change something, a highlighting component (6) shows John in-place annotations on diagram elements to keep him aware of Mark's in-progress changes.

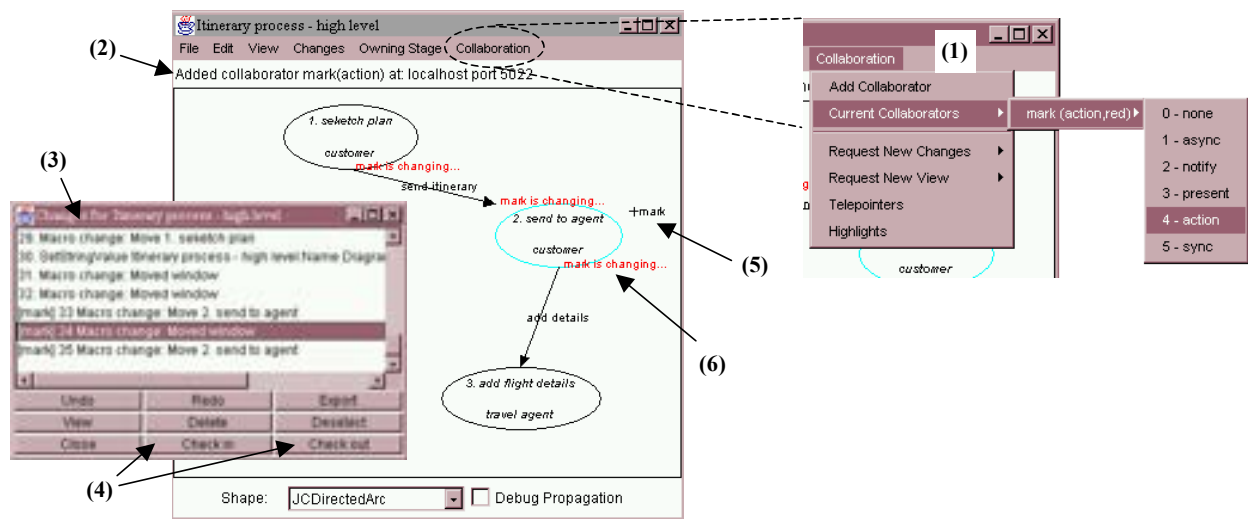


Figure 2. Examples of collaboration support groupware components in workflow tool.

Communications

Consider John and Mark collaborating to plan a trip. They share a travel itinerary editor (tree layout of trip details) and various visualisation views. One such high-level view is a map visualisation showing the legs of the travel (Figure 3). John and Mark may synchronously or asynchronously make changes to their shared travel plan. After changing the plan, John adds a note annotation on the map visualisation. This can later be clicked on and note read by Mark, John or other users (2).

If John and Mark are working asynchronously (one is off-line) they can communicate via email-style messages using an email component. If they are both on-line, they can use a text or audio chat (3) component to communicate and discuss their shared travel planning work. A text message component can be used to provide scrolling messages inside diagram and document windows, giving in-place messaging (4) as well as notification. Changes to the travel plan can be viewed (as in previous workflow example) with an event list component. The same component can be used by Mark to review a conversation with John (5), or by another user to view a conversation they missed.

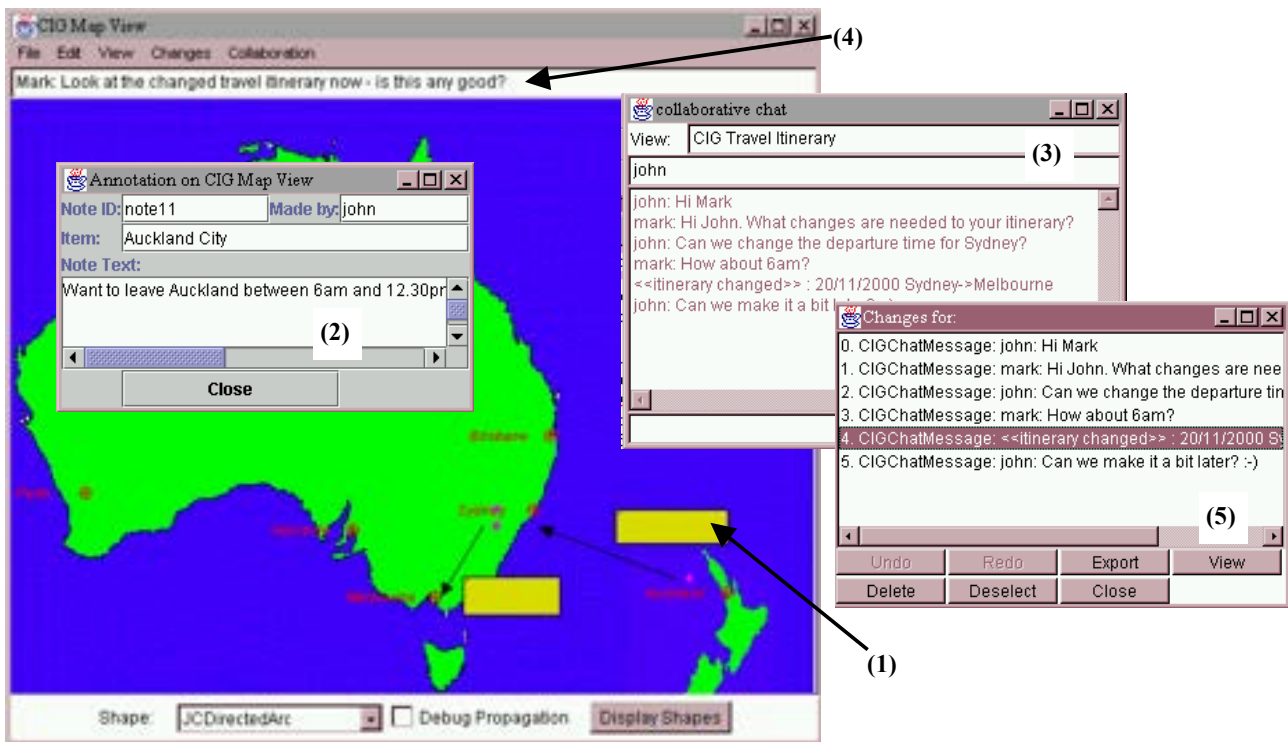


Figure 3. Examples of communications support groupware components in travel planner.

Co-ordination

Consider John and Mark editing a CASE diagram synchronously. At various times they want to be notified of changes that each other are performing, but may miss the transient awareness support described above (cursors and temporary annotations) e.g. because they briefly go to another window to do some work. A notifier component can be configured by Mark to inform him, in various ways, of changes John is making. For example, John may request a chat message be sent to him by the notifier when Mark makes a change (1) or a text message be sent (2). The notifier makes use of the chat and text messaging components to do this. In addition, when collaboratively editing the view, a locking component can explicitly deny access temporarily to components while they are being edited (3).

Because John may go off-line for some time, he may want annotations made to the document to inform him of Mark's changes. John may request e.g. a note annotation be automatically created (4) against changed items, or changed items highlighted when he goes back to this view (5). The notifier makes use of the note component and highlighter component to do this.

John and Mark wish to remain aware, not only of low-level work artefact modifications, but also higher-level tasks each is doing. A shared to-do list component is used to record tasks (6). The tasks can be manually entered by users, or obtained automatically from a workflow system like the one illustrated above. Task information can be displayed in work diagrams, for example using text messages or annotating the diagram. A to-do list component has annotated the CASE diagram window title (7) to indicate what task John is performing while modifying the diagram, allowing Mark to see this at a glance.

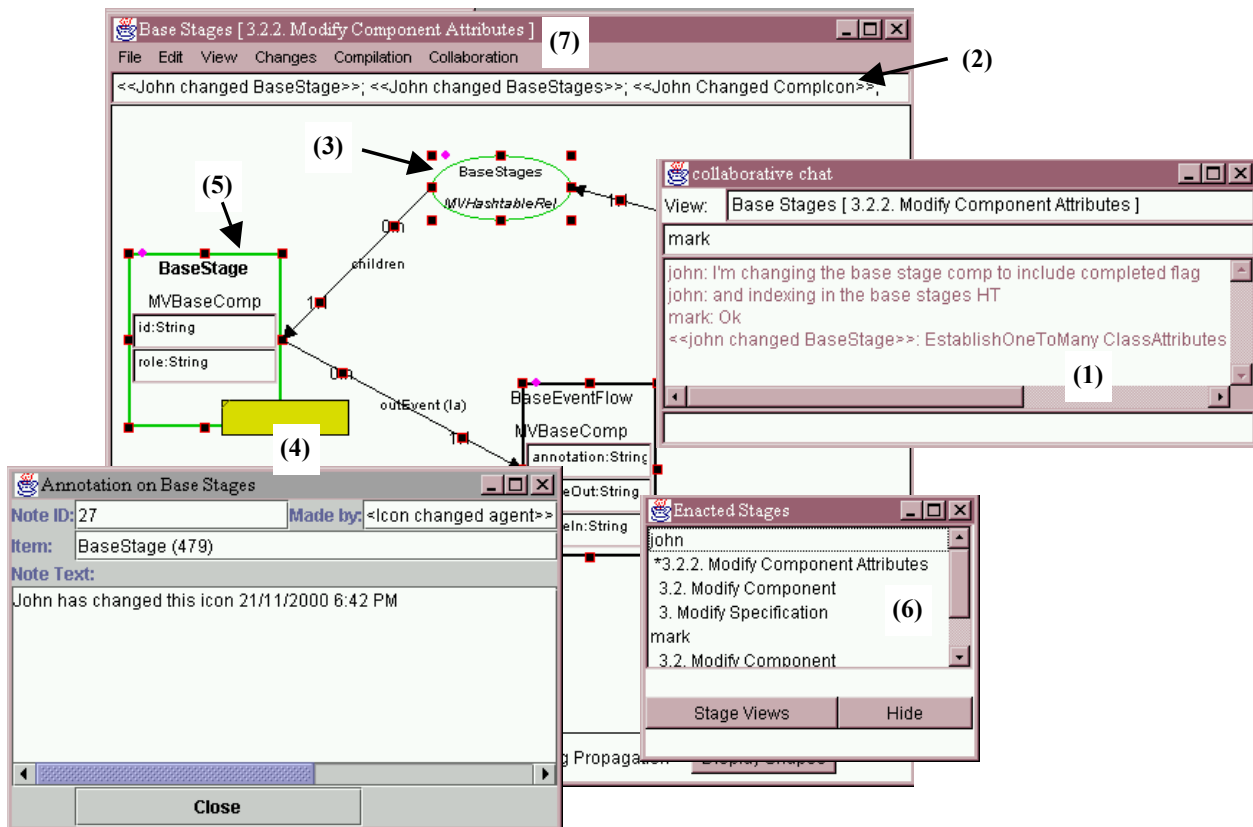


Figure 4. Examples of co-ordination support groupware components in CASE tool.

Plugging in Groupware Components

John and Mark need to plug groupware components into their applications in order to make use of the facilities described above. Figure 5 shows examples of the three ways they can do this. In (1), John opens a file containing groupware client code – when this is opened the component code in the file is loaded and the groupware component initialised. In (2), John uses a “Wizard” to deploy a selected groupware component from a set of available plug-in components (in this example, a notification component is being configured). These two approaches are appropriate for novice users as they hide details of the plug-in component architecture and links. In (3), John adds new components and connects them to existing components using a visual component deployment tool. This allows John to add in new groupware support from pre-existing components. This approach is suitable for expert users who wish to have flexible control over component creation and linkage.

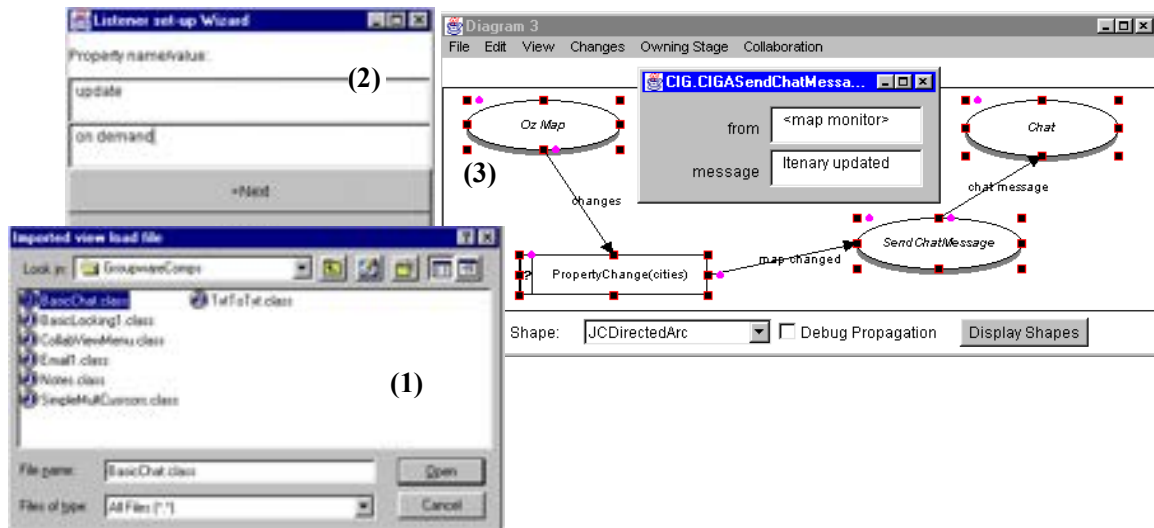


Figure 5. Examples of adding groupware components to an application.

When a groupware component is added to one of our applications, it may need to adapt parts of the interface the user of the application interacts with. For example, in Figure 2 additional menu items have been added to the workflow diagram menu by the collaborative editing support component when it was initialised. A versioning component added check in and check out buttons to the editing event history window. In Figure 3 the text messaging component added a message display field below the travel visualisation diagram's menu bar, and the email, note and chat components added menu items to the Collaboration menu to allow users to access their facilities. Such adaptations allow users to seamlessly interact with added groupware support. There are complex issues that arise when making such user interface adaptations that are beyond the scope of this paper to discuss in detail (e.g. What if two components want to add the same-named menu item? What if adding buttons changes the layout of a button panel? What if one component wants to disable a control and another assumes the control is accessible?). We discuss these issues in detail elsewhere¹⁸.

ABSTRACT GROUPWARE COMPONENTS

In this section we identify a number of abstractions we have identified when developing groupware components as illustrated in the previous section.

Taxonomy

Table 1 shows taxonomy of groupware components that we have found useful when developing such systems. Some components provide client-side user interfaces to support collaboration (editing, versioning), communication (notes, chat, messages) or co-ordination (locking, to-do lists). Some provide server-side centralised data and event management (message exchange, event exchange and message, event and version histories). Some provide infrastructure services (building data and event sending/receiving services, extensible user interfaces and persistency management services).

Component Category	Examples	Description	
Groupware clients	Collaboration	Multiple cursors	Shows other users' cursor positions
		Collaborative editing client	Provides configuration and collaborative editing facilities for application elements
		Versioning client	Provides version control facilities
	Communication	Chat client	Provides text-based chat between 2 or more users
		Email client	Email messages/documents
		Text message client	Scrolling text area in application's window frame
		Notes client	Note annotations
	Co-ordination	Locking client	Highlights items other users are modifying
		To-do list client	
		Notification client	Provides configuration interface for notifier
General-purpose	Event history client	Provides list of edit, message, note, to-do item etc events	
Groupware Servers (or "receivers" if Point-to-Point)	Message server	Used by chat, email, text message clients. Broadcasts messages to other users.	
	Data/event history server	Used by message server, note, to-do list and version clients. Stores list of retrievable messages/events.	
	Event server	Used by collaborative editing, notifier and multiple cursor clients. Broadcasts events to clients.	
Groupware Infrastructure Data/event exchange	Data/event sender	Encodes data and events for sending client->: server or server->client	
	Data/event receiver	Decodes data and events sent server->client or client->server	
Persistency Infrastructure	Database access	For storing large numbers of small, discrete, well-structured data items	
	File Access	For storing smaller numbers of unstructured data items	
	XML Access	For storing moderate numbers of semi-structured data items	
User Interface Components	Extensible menus, button panels, text areas, frames etc	Range of components allowing user interface building and run-time adaptation	
Domain-specific components	Itinerary Editor/Server CASE diagram editor Workflow diagram editor Workflow engine ...	Application components plugging groupware into	

Table 1. A basic groupware component taxonomy.

Architectures

Groupware can be built using two fundamental architectural approaches: client-server and peer-to-peer^{1, 19}, as illustrated in Figure 6. Client-server has the advantage of simplicity and centralised data management, but the disadvantage of single point-of-failure and bottleneck in the server. Peer-to-peer has the advantage of robustness (no single server to fail) and flexibility, but is generally more complex to build with synchronising copied data often difficult. Our groupware components are designed and implemented to operate in either mode, and can also utilise a "hybrid" approach of utilising both client-server and peer-to-peer styles. If operating in peer-to-peer mode, groupware client components communicate directly with other groupware clients. If operating in client-server mode, clients communicate with a server that broadcasts events and data between clients as necessary. All our groupware components by default hold copies of shared data allowing peer-to-peer operation, but can use a server to synchronise data or provide a single, central data/event distribution point.

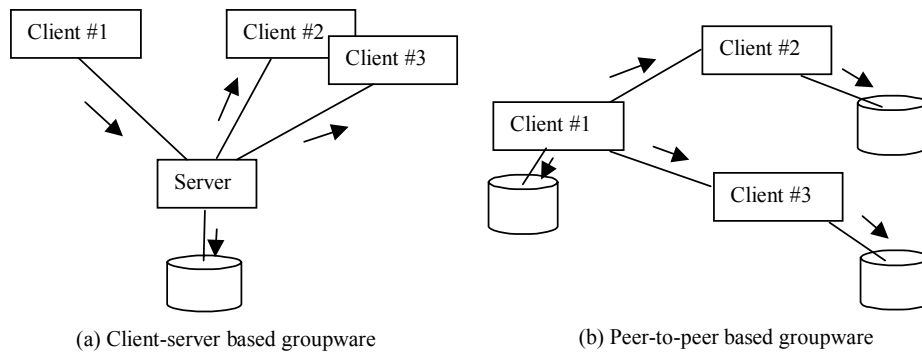


Figure 6. Client-server vs. peer-to-peer vs. hybrid groupware architectures.

Figure 7 illustrates the basic structure of our component-based groupware systems. Groupware client components share a common set of user interface and infrastructure components. They may also share event and local persistency (for peer-to-peer groupware) management components. Client components interact with domain-specific system components, like the travel itinerary editor and map, workflow system diagram editor and CASE tool diagram editor, detecting events and applying updates to the domain-specific interface. Clients communicate either with one or more groupware servers (if client-server operation) or one or more other groupware clients (if peer-to-peer operation). If operating in client-server mode, clients usually use the servers to store and retrieve groupware data. If operating in peer-to-peer mode, clients also store their own data locally.

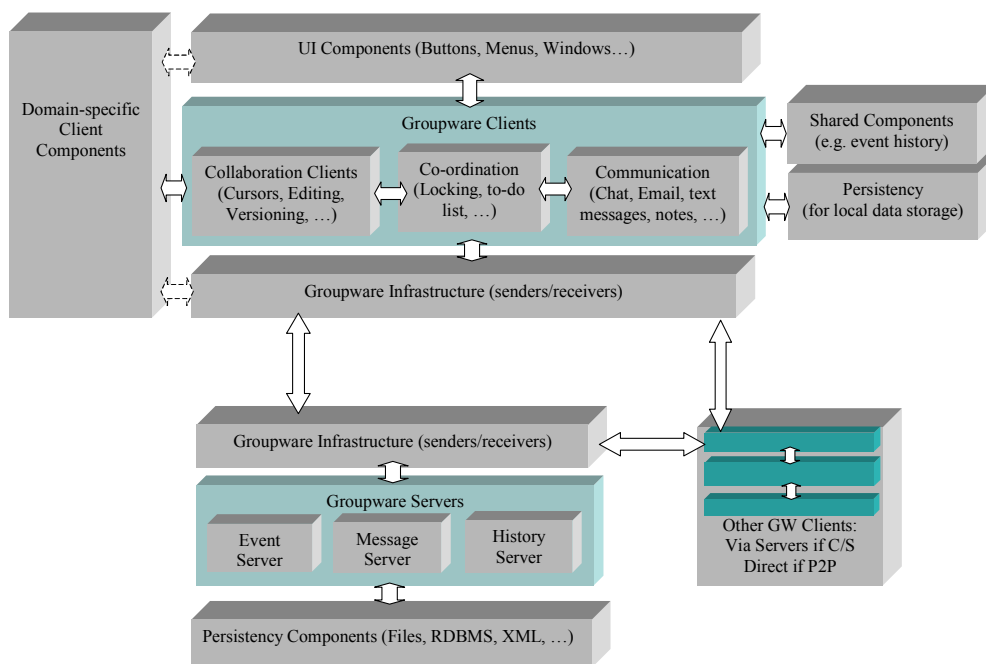


Figure 7. Basic groupware component infrastructure.

Collaborative Editing

To illustrate at a high-level how the groupware examples from the previous section are built using these groupware component abstractions we will consider three scenarios: collaborative editing and group awareness (via multiple cursors); editing event notification using text messaging and note annotation; and adding a new groupware component that must integrate itself with both domain-specific components and other groupware components.

Consider John and Mark collaboratively editing a workflow diagram, as illustrated in Figure 2. Figure 8 shows how the various components interact when a) John moves his cursor (keeping Mark

synchronously informed of John’s workspace interactions), and b) John modifies a workflow icon (showing Mark the changes made). These examples both use a client-server architectural approach to connect the distributed groupware components. In the top example, John moves his cursor. The Multiple Cursor Awareness groupware component is informed of this event after the cursor has been moved, and sends a “mouse moved” event to the Event Server (using an Event Sender component). The Event Server broadcasts this to all interested users’ multiple cursor components that have previously subscribed to other users’ mouse move events. Mark’s Multiple Cursor component receives notification of John’s mouse move and displays a “cursor” for John in Mark’s corresponding workflow diagram.

In the bottom example, John changes the name of a workflow stage. This results in the edit event(s) being sent to the event server. If fully synchronous editing is being done, the event server first checks if the data John is editing is “locked” i.e. being changed by another user. If not, it is locked for John’s use and the edit event is stored by the history server and sent to others editing the same diagram. Mark’s collaborative editing component receives the event(s) and applies updates to Mark’s diagram. This can be done automatically or the events first presented to Mark for approval. John’s Collaborative Editing component records the editing event(s) locally so they can be undone. Stored editing events can be exchanged as a block with another user to support asynchronous work i.e. the other users merges the stored changes with their own version of the diagram.

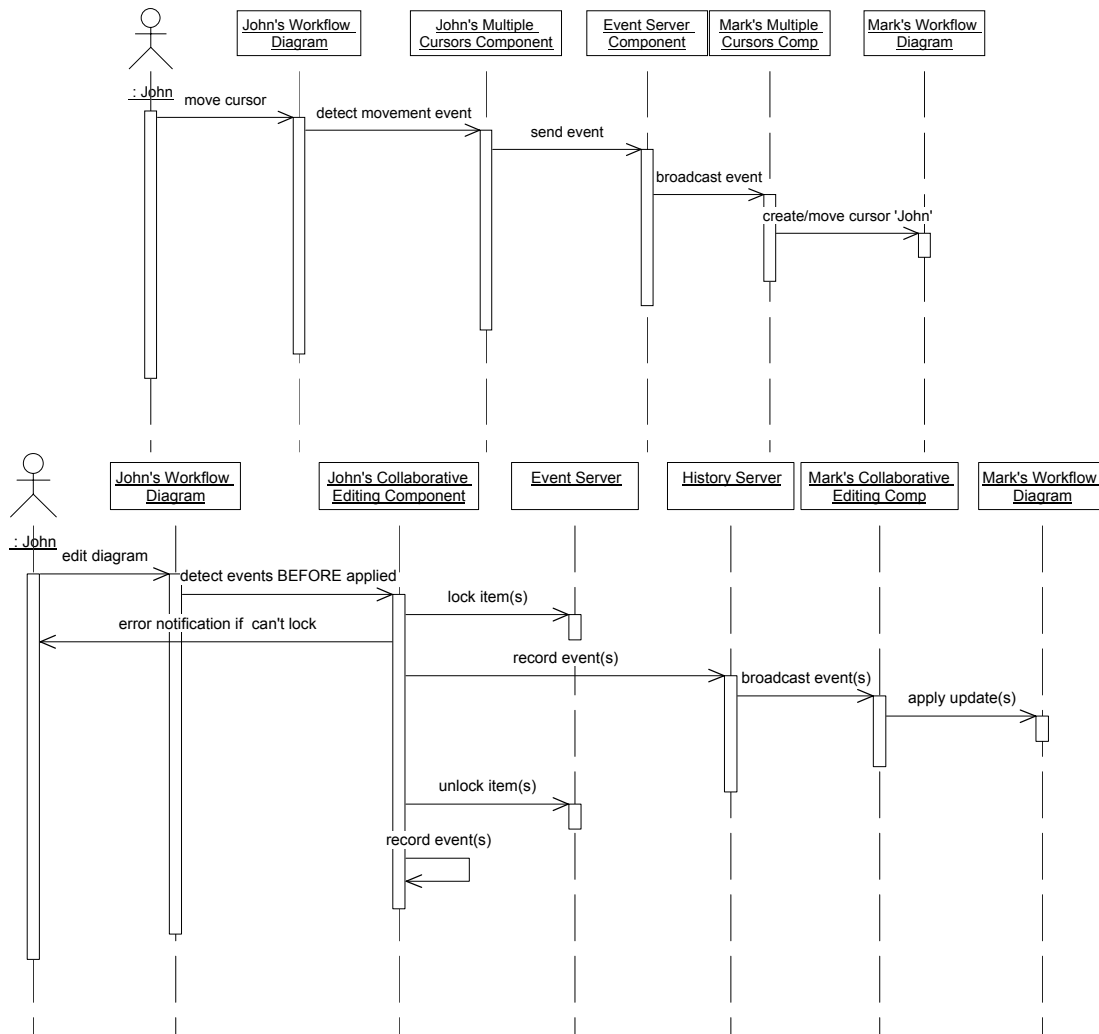


Figure 8. Example groupware component interactions: a) group awareness and b) collaborative editing.

Co-ordination

Consider John and Mark collaboratively planning a trip, as illustrated in Figure 3. When John changes a travel item Mark needs to be informed. This can be done synchronously via e.g. a text or chat message, or asynchronously via e.g. an email message or note annotation on the itinerary view. In the example in Figure 9 notification is done by a notification component using two communication components: a text message appears on Mark's screen and a note annotation is added to the changed item. These examples use a peer-to-peer architecture i.e. John's notification client communicates directly with Mark's message and annotation clients. As with the collaborative editing example above, a history component could be used to remember the editing or text message events.

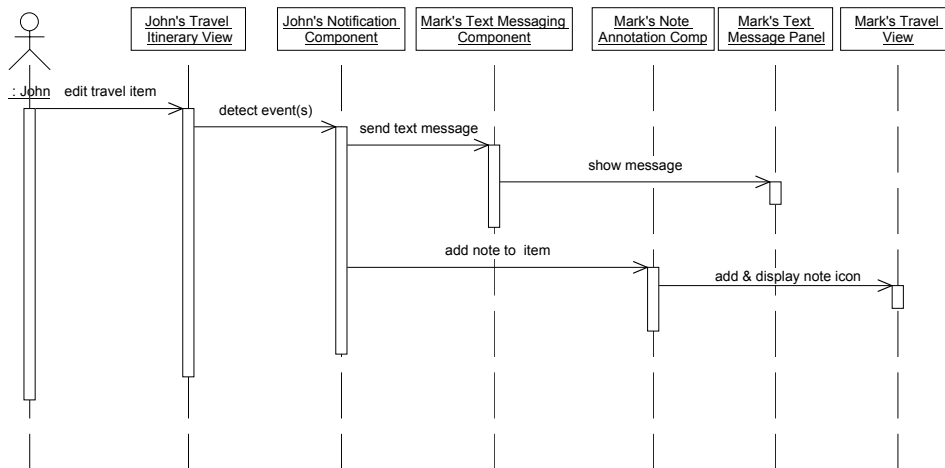


Figure 9. Example interactions: edit event notification via messaging and note annotation.

Plugging in Components

Consider John wanting to collaboratively edit a CASE tool diagram with Mark. John needs to add collaborative editing functionality at run-time i.e. the CASE tool doesn't currently have such a feature. Figure 10 illustrates the basic component interactions that take place. John chooses a collaborative editing groupware component from a list of available plug-in components for his CASE diagram. The collaborative editing component discovers the user interface elements the diagram provides and adds a set of menu items to the CASE diagram view, allowing John to configure collaborative editing facilities. The collaborative editing component then determines the events the diagram generates and the event sender available to the diagram for it to use. Finally, the collaborative editing component registers itself with an event server (or other users' environments directly, if a peer-to-peer architecture). John can then set the "level" of collaboration with Mark (e.g. asynchronous, synchronous or semi-synchronous) via the new menu items.

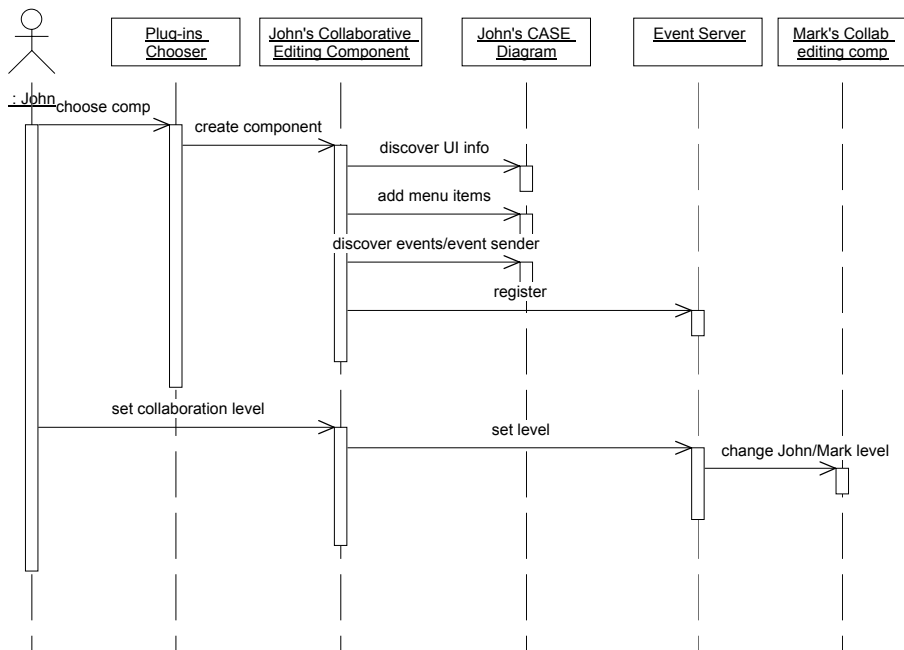


Figure 10. Example of adding a collaborative editing groupware component.

JVIEWS FRAMEWORK

Framework Abstractions

In order to build flexible groupware components like those outlined in the previous section a developer needs a range of abstractions with which to construct such components. One approach is to make use of a component-based class framework that encapsulates the fundamental building blocks for both the groupware and domain-specific components. Key features of such a framework should include:

- Components and inter-component relationships. These provide pluggable encapsulations of data and functionality, and manage inter-component relationship management.
- Component update events. These are generated when component or inter-component relationships are modified, or key events relating to the component's state occur.
- Before- and after- subscription to events. Components need to be able to be informed of changes to other components both after these have been made, but also BEFORE they are changed (e.g. to support locking and checking of shared components). JViews also allows components to receive notification of events being sent between two other components, enabling over-riding of their default event notification behaviour without code changes.
- Description of component functional and non-functional characteristics. The user interface, distribution and persistency management approaches used by components must be inspectable by other components so run-time adaptation and reuse of components can be adequately supported.

We have developed a framework called JViews, originally designed for building multiple-view, multiple-user design environments e.g. CASE tools, CAD tools, programming tools, workflow tools etc^{19,17}. JViews incorporates component-based building blocks for constructing such systems from reusable parts. When developing JViews, we originally added groupware capabilities to the framework classes i.e. hard-coded them in, in a similar fashion as done with GroupKit, COAST and Meta-MOOSE^{9,7,8}. Thus while we used reusable component abstractions to built applications, our groupware support followed the "monolithic" application development approach, where developers had little control on what groupware support was included in specialised components, and this

resulted in sometimes inappropriately reused and over-complex component functionality^{19, 18}. In addition, users were not given any control over what groupware functionality their environments supported, even if this was inappropriate to their specific needs.

Figure 11 illustrates the key abstractions in our JViews-based component framework. Components have attributes (state) and specialisations have operations for modifying state. Inter-component relationships may be simple zero-to-many links (component-to-component) or component-relationship component-component. Relationship components include hashtables (indexed) and vectors (sequential, unindexed). Events describe component state changes and a number of specialisations exist, including attribute and relationship changes. Aspects describe characteristics of the component e.g. user interface, distribution and persistency mechanisms provided or required.

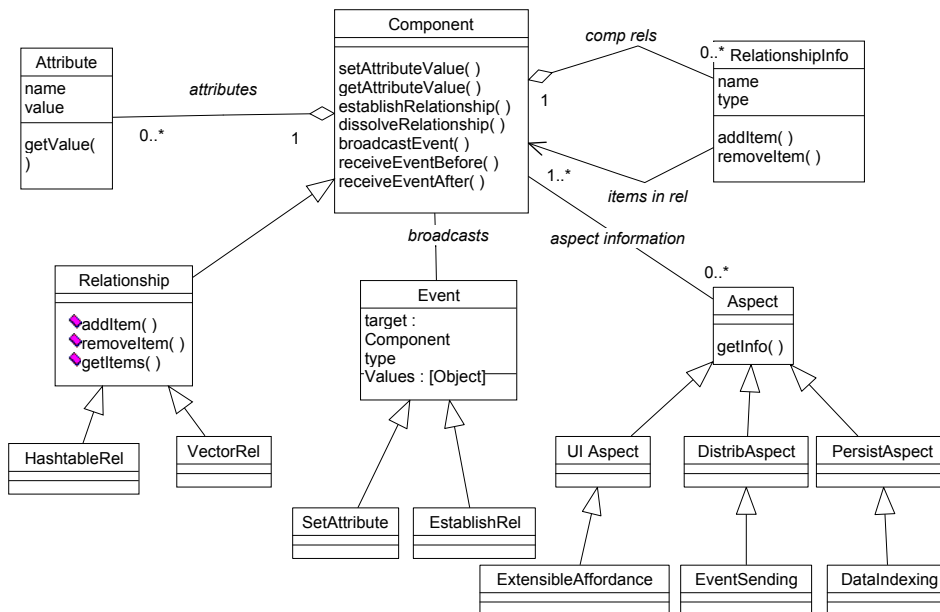


Figure 11. The JViews component-based framework.

The following table outlines some of the specialisations of these component, event and aspect framework classes.

JViews Abstraction	Examples	Description
Components	Base Components	Repository-level informational component. Used to model state shared between views in design environments.
	View Components	View-of-base (copy) – icon, connector, group, text etc. Used to model different views of the base components.
	Presentation Components	UI component e.g. window, panel, button, icon, connector, layout constraint etc.
	Relationship Components	Manages bi-direction connectivity between components. 1:1, 1:n, m:n, ordered (e.g. vector) and unordered, indexed (e.g. hashtable, B-tree) etc.
	Infrastructure components	Event/data sending/receiving; persistency management; 3 rd party component integration
Events	Component Created	Component added
	Component Deleted	Component deleted – this is REVERSABLE in JViews – component not thrown away, just relationships to others dissolved (and can be undone)
	Attribute Modified	Set value (records before and after values of attribute so can be reversed easily)
	Established relationship	Components connected
	Dissolved relationship	Components disconnected
	UI Events	Window opened, button clicked, mouse moved etc
	Macro Events	Group of events – can be undone/redone/transported as a group
Aspects	UI Aspects	Frame, extensible panel, extensible menu, can be disabled/hidden, UI component information (size, colour etc), how display properties etc
	Distribution Aspects	Sends/receives data, transports data, encrypts/decrypts data, location information, performance information
	Persistency Aspects	Saves/loads data, produces data, storage mechanism used, query support
	Configuration Aspects	Software interfaces for configuration, UI for configuration, configuration properties

Table 2. Examples of JViews component, event and aspect abstraction types.

Collaborative Editing Components

Consider again the example of John and Mark collaboratively editing a workflow diagram. Figure 12 shows how this collaborative editing scenario, here using a client-server architecture, is realised using Jviews component abstractions. John's collaborative editing component subscribes to workflow diagram update events (1). When John changes a workflow diagram by direct manipulation (2), one or more events are created to describe the view state change about to be applied (3). These events are sent to the collaborative editing component (4) which then sends them across the network via the event sender (5) to the event server (6), which processes these events (7). The event server checks if the item(s) about to be changed by John are already locked e.g. Mark is editing them (8) and a response returned to John's collaborative editing component (9). If locked, this returns an error event to John's collaborative editing component which then aborts the attempted editing change(s) (10) and informs John of the concurrent editing clash. If unlocked, the event server locks the item(s) and sends the edit events to the history server (11). The history server component notifies all other users synchronously editing this view (12), and in this example Mark's collaborative editing component is informed (13), which then updates Mark's workflow diagram (14). Finally, John's collaborative editing component records the edit event(s) (15).

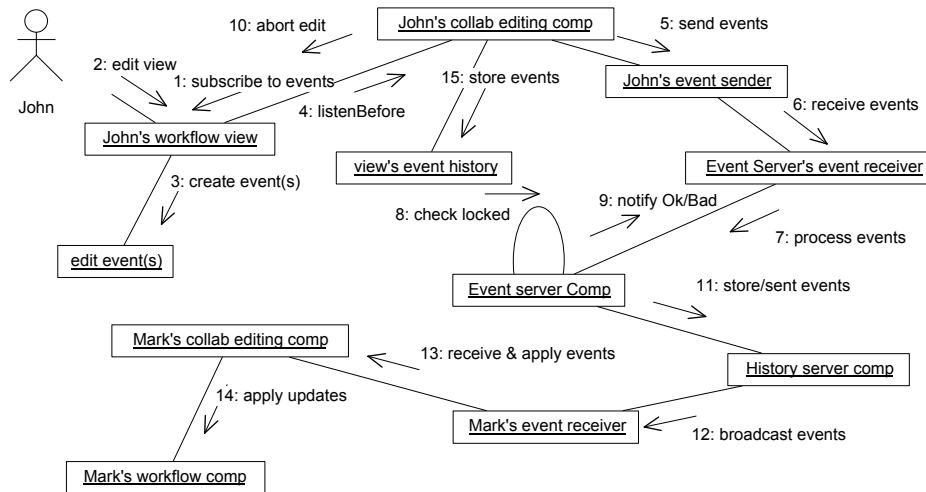


Figure 12. Example: JViews event handling for collaborative editing.

JViews uses the event sender and receiver components to achieve event distribution across a network (we have implemented socket, Java RMI and CORBA data transportation protocols for such components). In some situations the sending JViews application broadcasts the events whenever they occur, while in others filters are applied to the events to determine whether or not to transport them across the network. For example, to support multiple cursors the sending JViews environment records cursor movement events but only sends a single, absolute position of the user's cursor every second (by default – this can be changed by user preferences). This significantly reduces network overhead. Similarly, a notification component is told of all changes made to an application component, but we apply filters at the source and only sends events matching specified criteria to receiving components across the network. While such constraints can be specified in either source or target JViews application, sending many irrelevant events to the target system is costly.

Co-ordination Components

The implementation of a notification scenario is illustrated in Figure 13. In this example certain changes made by John to travel itinerary items e.g. item creation, arrive/depart date change etc, are indicated to Mark via synchronous text messaging and asynchronous note annotation. John's notification component subscribes to the events generated by the itinerary view (1). When John changes an itinerary item (2), events are created to describe this (3) and sent to the notification component after the change has been made (4). John's notification component then determines if the event meets the criteria describing the kind of change the user has specified. Send message/add note events are then created (5) and sent to Mark's note annotation and text messaging components (6, 7, 8, 9), instructing them to display a text message and annotate Mark's travel item view. When these events are sent to Mark's components the text messaging component formulates a new text message (8, 10) and displays this (12). Mark's note annotation component creates a new note (9, 11) and displays this in Mark's travel view (13) and records the note details for future perusal (14).

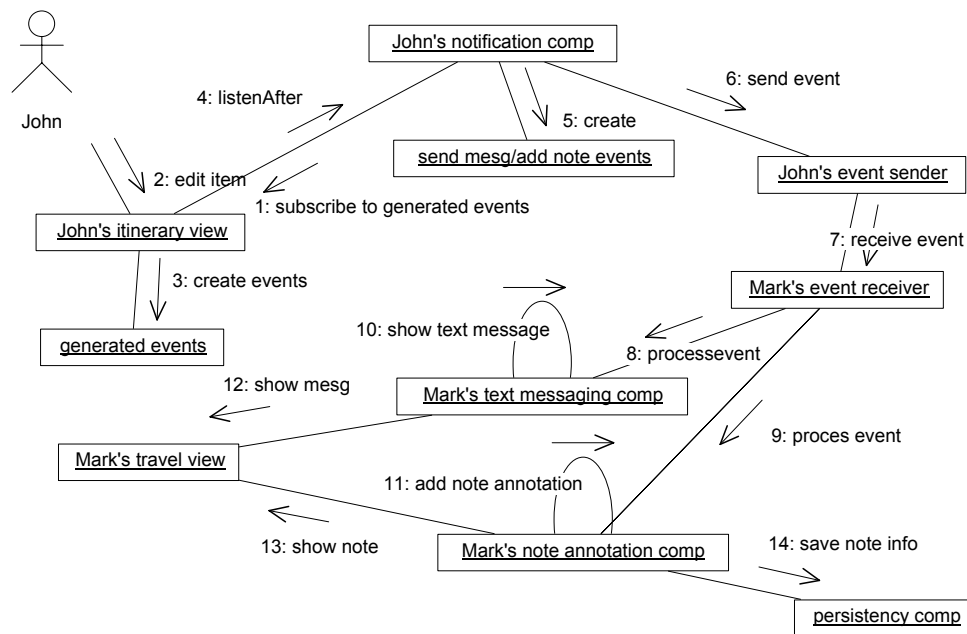


Figure 13. Example: JViews-implemented groupware notification and awareness components.

Plugging in Components

Our groupware components built using Jviews utilise “aspect” information encoding associated with other Jviews components to determine, at run-time, how to extend related component user interfaces, determine events to subscribe to, and to reuse related component distribution and persistency services³⁷. To illustrate how this works, Figure 14 shows some interactions between a collaborative editing component and workflow diagram component when the collaborative editing component is initialised.

When the collaborative editing component is associated with e.g. a workflow diagram, it determines the available UI aspects of the diagram’s component (1), so that it can extend the diagram’s user interface (2) to add the collaborative editing configuration interface items it requires (3). In this example the collaborative editing component adds “affordances” (e.g. Add User, Set Collaboration Level, Send View, etc) by calling functions published by the workflow diagram’s extensible menu aspect information (2). The extensible UI aspect object provided by the workflow diagram creates menu items to provide the required affordances (3) and adds these to the workflow diagram main menu bar (4). The collaborative editing component has no direct knowledge of where the items are added nor even that they are menu items (they could be implemented as pop-up menu items or buttons). Similarly, the collaborative editing component subscribes to events the workflow diagram generates (5-7) by introspecting the event generation aspect object the workflow diagram provides. It also determines and reuses the local event sender and persistency components used by the workflow diagram (8, 9) via its distributor and storage aspects. If the collaborative editing component is associated with a different diagram e.g. the travel planner map visualisation, then it will obtain the same kinds of aspect information but using these may provide quite different results e.g. pop-up menu items are added, a different set of events are subscribed to, and different distribution and persistency components are obtained.

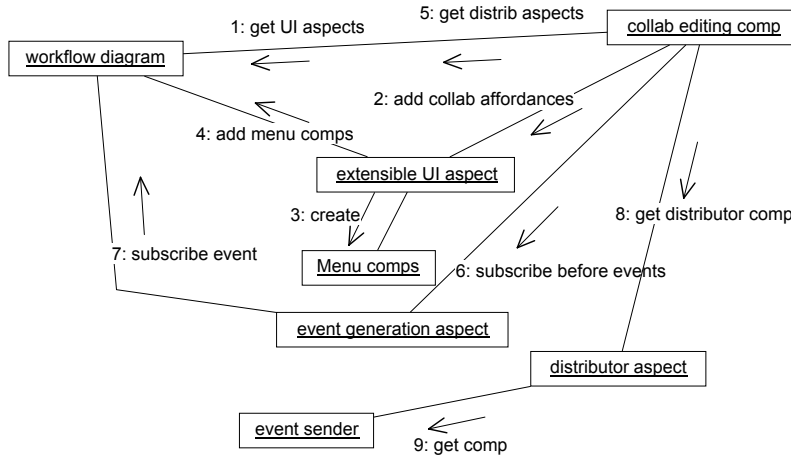


Figure 14. Example: adding groupware components and using JViews component aspects.

DESIGN AND IMPLEMENTATION

JViews Event Model

In order to implement the groupware components described in the previous sections, we need to have a component framework that provides sufficiently flexible run-time linking of components and handling of generated events. Jviews allows components to be linked by name at run-time without any compile-time knowledge of one another. We use a “relationship information” (RelInfo) class to do this. This allows any JViews component to be linked to one or more other Jviews components by a named relationship, and for the relationship to be queried by name. This inter-relating mechanism is also used to support flexible event subscription between Jviews components. Each named relationship link indicates when the related components should be told about events affecting a component. Relationships in Jviews are bi-directional so either end can listen for events affecting the other end of the relationship. Relationships can be named and used statically i.e. at compile-time, but for our groupware components most are constructed at run-time i.e. are dynamic. This means groupware components can establish relationships with domain-specific components, each other or infrastructural components, and none require any compile-time knowledge of each other.

Each Jviews component maintains a list of related components. This is a bi-directional relationship where each related component also maintains a reverse link to the component. When a component is deleted, all related components are informed of this so no “dangling links” are possible. Deletion of components can also be undone as the deleted component’s relationship links are retained - only related component’s links are updated to remove reference to the “deleted” component. The relationship link structure is illustrated in Figure 15 (a).

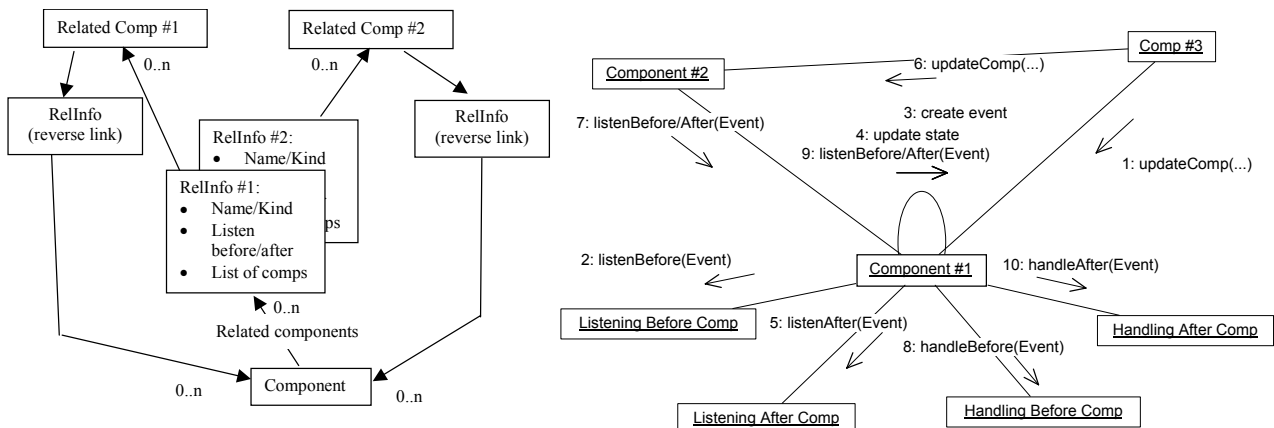


Figure 15. (a) Jviews relationship management and (b) flexible subscribe/notify.

Each Jviews RelInfo relationship-implementing object records when to tell all of the components it relates about state changes to the JViews component they belong to i.e. when to send events to them. To subscribe to events generated by another JViews component, a component must establish a relationship to that component (if none already exists) and then specify for that relationship when it wishes to receive the generated events. Events can be sent to related components:

1. before a component's state is actually changed i.e. "listen before changed";
2. after a component's state has been changed i.e. "listen after changed";
3. when a component has received an event from a 3rd component but before the component has handled the event itself i.e. "handle change before"; and
4. when a component has received an event from a 3rd component and the component has handled the event itself i.e. "handle change after".

Event Propagation

This subscription/notification mechanism allows components to monitor other components state changes before or after the changes have been made (1 and 2), as well as intercept events sent from one component to another before or after the receiver has handled the changes (3 and 4). Note that when a component specifies it wants to receive events from another JViews component, it receives ALL generated events i.e. there is no subscribe-to-specific-events in the basic JViews subscribe/notify model Figure 15 (b) illustrates this subscribe/notify mechanism in Jviews.

Other event models, such as those provided by JavaBean event listeners³⁹, Model-View-Controller architectures⁴⁰, and systems like Rendezvous' ALV⁵ generally provide only "listen after changed" event subscription. This makes it very difficult to implement many groupware facilities like locking, some kinds of notification and versioning without application code changes to support these⁴¹. Some event models support listen-before notification, such as the ItemList⁴² and ODGs⁴³. The Jviews model is more flexible in that monitoring of components receiving events is also supported, allowing useful over-riding of event handling behaviour to be seamlessly added without the knowledge of the sender or receiver relationship structures. While Jviews doesn't provide subscription to specific kinds of events nor prioritisation of event notification directly, we have implemented reusable components that provide event filtering and priority-ordered distribution of notification.

Figure 16 illustrates the algorithm used to propagate events between related Jviews components. When updated, a Jviews component creates an event object to represent the change and then broadcasts this event to all components linked to it whose RelInfo objects indicate the related components want to be sent the event before the state change is done i.e. "listen before" subscription. We use this mechanism to implement locking and highlighting for our groupware components. If these components have "handle before" listening components i.e. ones that want to process the event before the listen before components, these components are sent the event to respond to first. This is used to implement some forms of collaborative editing and multiple cursors where these groupware components listen indirectly to event stores and window frame components, sent changes by view components. Responding components may throw an exception, causing the event to be aborted and not actioned, or return a null event object indicating event propagation should go no further (i.e. "consume" the event). Normally, the state change described by the event is then applied by the broadcasting component to itself. After the state change has been done, the component sends the event to all "listen after" components for them to respond to it. This is used to implement chat and email messaging, to-do list item updating and version exchange in our groupware components. Each listen after component may send the event to "handle after"

components once they have processed it. This is used to support recording of editing, message and co-ordination events in our groupware components.

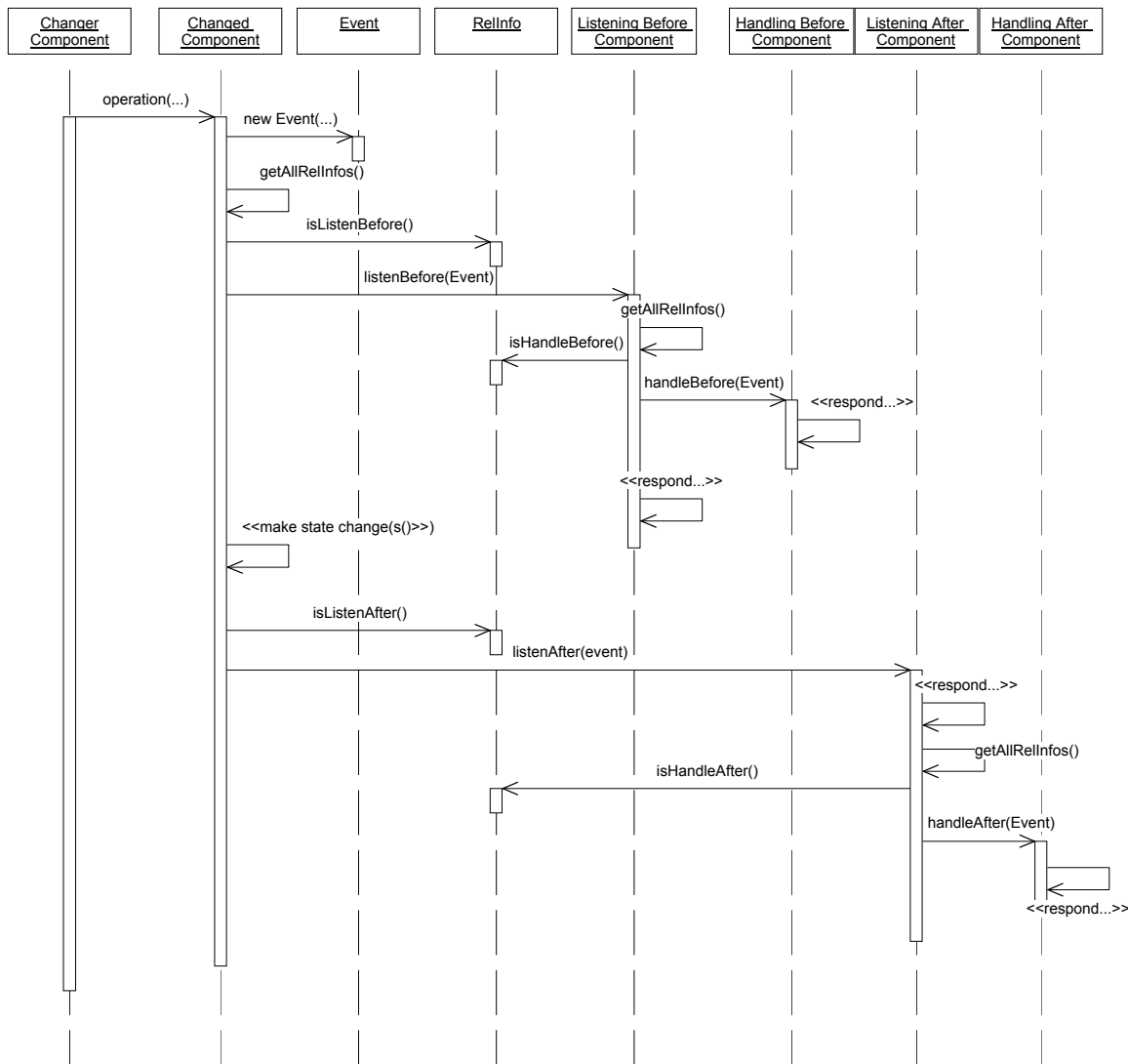


Figure 16. Event propagation in JViews.

We have developed several reusable components supporting the distribution of JViews events across a network and more refined event subscription mechanisms. Event distribution is done by sender/receiver components. The sender subscribes to Jviews component events and broadcasts these to one or more remote receivers (JViews component state transfer is supported by the same mechanism). A cross-application unique ID generation mechanism and component versioning are provided to allow multiple JViews applications to simultaneously maintain different versions of the same JViews components. This is needed for peer-to-peer architectures¹⁷. As noted in Section 5, care is needed when setting up such remote event broadcasting infrastructure to ensure wherever possible event filtering is done at source to minimise unnecessary event transport across a network.

Implementation Technologies

We have implemented our groupware components using several technologies. Our JViews component implementation framework is implemented in Java, and JViews components extend the JavaBeans component model. We added our extended event subscription/notification model to JavaBeans, together with our aspects codification technique and a number of abstractions to support multi-view tool development. We also developed a range of JViews components to provide reusable view editing tools, event histories, event and component distribution and persistency management.

These provide developers basic component building blocks to realise the applications and groupware components described previously.

Our distributed data and event management components supply developers with various abstractions for constructing distributed applications, specifically our groupware components. Originally we developed event and data broadcasting components using TCP/IP sockets, as provided by Java's APIs. We implemented a custom component and event data serialisation mechanism to marshal and demarshal JViews components and events between distributed applications. This used a textual data encoding and used parsing to decode received data. We also developed a custom remote component identification (naming) mechanism and custom component versioning support. We used point-to-point and client-server based application interaction, with an emphasis on point-to-point communications to improve application robustness and performance. These custom naming, serialisation and versioning techniques were necessary in order to support functions such as the replication-based flexible collaborative editing of views, identification of remote task automation and workflow agents, and maintenance of multiple versions of event and message histories. Our TCP/IP socket-based model works well in many application domains and for many of our groupware components. However, the text-based data encoding is quite slow and inefficient for high volume event inter-change. It also takes much of programmer effort to use the (very low level) socket-based distribution components, and this is error-prone and difficult to scale and extend. The custom naming and serialisation mechanisms make our components very difficult to combine with third-part tools, particularly for task automation.

We have retained our socket-based distribution management components for remote component versioning and multiple component data exchange. To support event exchange (e.g. multiple cursors, locking, messaging and remote workflow and notification event broadcasting) we have developed new distribution components using Java RMI and CORBA. The RMI-based event broadcasting components provide high-performance event subscription and notification facilities. They also support broadcasting a wide range of Java types far more easily than our custom socket-based protocol, allow improved remote component identification and location, and are easier for component developers to use. We use a modified form of our event serialisation and component identification mechanisms to encode component references, but let RMI encode all basic Java types and API objects. Our RMI event distribution components perform much better than our socket-based components for components with high-volume event exchange or for events with complex Java data types. They do not, however, offer any improvements in fault-tolerance and third-party component integration over the socket-based protocol.

We developed an additional component and event distribution facility using CORBA-based remote object interfaces to address these issues. Some components can be named, looked up and have their methods invoked remotely using these CORBA-based facilities. Occasionally we wanted to directly interact with third-party (i.e. non-JViews implemented components) components, particularly to support task automation and notification, and this technique is more general than using a custom socket protocol or RMI, and less work than building JViews component "wrappers" around third party remote components. There are also times when our groupware components want a more robust distributed method invocation infrastructure e.g. important notification agents and components supporting distributed component locking and version control. Visibroker™, the CORBA implementation we used, offers much better support for multiple remote object instantiation and transparent fault tolerance than our socket-based and RMI-based mechanisms.

An advantage of our textual component and event encoding is that data serialised using this facility can be used to make components persistent in indexed files. We originally provided persistency management components that stored component and events locally in serial or indexed files, or remotely via a shared file server. Like our socket-based component and data distribution

mechanism, this approach unfortunately proved difficult to program and not scalable for large applications. We have recently used an object database (PSE Pro™, a simplified form of the ObjectStore™ OO database) to improve persistency management. This provides a high performance local or remote component and event storage facility. We also extended JViews components to enable basic component versioning to be supported using the object database. One disadvantage is that while programmers could previously simply attach a persistency component dynamically at run-time to any JViews component and have it transparently stored and retrieved from a file, PSE Pro™ requires post-compilation and annotation of Java binary files. This means any JViews components to be made persistent with our PSE Pro™-based persistency components need this post-compilation annotation run on them before any instances have been created. This limits dynamic choice of a persistency mechanism by our groupware components.

EXPERIENCE

We have built a range of collaboration, communication and co-ordination groupware using our component-based approach. Collaboration groupware includes collaborative editing, including both synchronous and asynchronous editing support, multiple cursors, group awareness messages, and version control. Communication groupware includes text chat, email messages, scrolling text messages, and note annotations. Co-ordination groupware includes item locking, shared to-do list and workflow-based co-ordination and task automation. We have deployed these groupware components in a workflow system, a CASE tool, a travel planner, a software architecture design environment, a software component development tool, and a distributed system application generator. These groupware can be used independently or in groups, and all can be plugged into or removed from a tool at run-time. We have developed both peer-to-peer and client-server based versions of most of these groupware components.

Performance of the groupware implementations we have developed to date varies depending on the technology used, the architectural style of the groupware organisation and the number and locality of users. Considering response time, all of our groupware perform well when a small number of users on a local area network are being supported. If more than a dozen users are concurrently using groupware facilities, response time degrades due to bottlenecking of the shared servers. This can be partially overcome using peer-to-peer networking, but memory and local disk storage go up considerably, and each client utilises a socket connection for every active collaborator, which can exhaust available connections on some host machine configurations. Some groupware functions well over wide-area networks, such as low-bandwidth chat, email, note annotation, to-do lists and workflow events. Synchronous collaborative editing, text messaging and multiple cursors work across a modem connection but perform quite slowly i.e. a considerable time-lag can be experienced by the users. This is due to before/after event subscription for synchronous editing, and large numbers of small event sends for multiple cursors and automatically generated text messages.

While our JViews event mechanism and groupware components generate potentially many events, there may be ways to mitigate large numbers of network event broadcasting. These include using “multiplexing” of event sends i.e. having sender components group all (unrelated) events generated in a specific timeframe and forward as one group to the receiver(s) on other machines. Similarly, some of our event generating components, like the mouse movement events, are aggregated into a single event (e.g. one per second) and only this one event is broadcast. We plan to investigate giving users some control over such approaches, allowing them to tune aspects of event broadcasting to enhance application performance.

We prefer using a peer-to-peer architecture for our groupware as this is more reliable than the client-server variant: with all of our groupware facilities, a client can fail at any point and other collaborating users can continue to work unaffected (well – without being able to communicate and

collaborate with the failed client until it reconnects to the network). Synchronising shared data in peer-to-peer groupware requires support for multiple versions of data, version merging and conflict resolution. This complicates the implementation of some groupware facilities, particularly collaborative editing, to-do lists and note annotations. Our JViews component model provides these facilities but at quite a low level of abstraction (a detailed discussion of JViews component versioning can be found elsewhere¹⁹). We have found the client-server versions easier to implement and maintain, though at a cost of less reliable facilities and sometimes-poorer response times for users.

None of our groupware currently supports security protocols, apart from simple authentication to identify each user and simple access control lists for groupware administration. If deployed in situations where secure data access and transmission is required, encryption support needs to be added along with secure control lists to ensure proper authorised access to each groupware component's facilities and shared data. We have prototyped some "security" groupware components that encrypt/decrypt transmitted data using CORBA object wrappers, and some simple access control lists for some groupware clients (for to-do list item access/update control and version access/update/deletion control). These all require further enhancement for larger scale application deployment. Interaction of domain-specific application security approaches and reusable groupware needs to be investigated, and handled in a similar way to our persistency and distribution component sharing via aspects.

Our groupware components have been evaluated as parts of two usability studies, one focusing on using our workflow system in a software development setting¹⁷ and one on our collaborative travel planner¹⁸. Both of these studies had small groups of users carrying out collaborative tasks with the workflow and travel planner systems and groupware facilities. The main usability problems reported from these studies include partial use of AWT-based Java user interface elements and modal-based diagram editing within our design environments. While these are not groupware component-specific issues, they adversely impact on the usability of these components within these applications. In general, users found the range of groupware facilities suitable and the basic functionality intuitive. The collaborative editing facilities have proven to have very good flexibility, though they take some time to become accustomed to⁴⁴. Adding groupware components to environments is difficult for non-expert users, requiring the provision of a more effective, easy-to-use end user component deployment tool.

We are extending our work with component-based groupware in a number of ways. We have been developing new techniques for adaptive user interface support we hope to use with our groupware components to improve the ability of developers to build adaptable interfaces. We have extended our aspect-oriented development approaches to provide richer descriptions of components using XML-encoded information, which we plan to use to describe our groupware and provide more automated run-time initialisation of components. We have recently been building thin-client groupware component prototypes. These use HTML- and WML-based user interfaces with all groupware functionality in web server components (realised by Java Server Page, CORBA and Enterprise Java Bean implementations). The integration of the thick-client groupware described in this paper and these thin-client groupware applications is an area of keen interest. We are looking at re-designing our server-side components to utilise the Enterprise Java Bean architecture and technology to increase their reusability and run-time deployment capabilities. We are developing a new infrastructure for event-based systems using SOAP-style web services and asynchronous messaging protocols, and plan to experiment with re-implementing some of our groupware communications using these architectures and technologies. This will hopefully provide more generalised interfaces to our groupware components and allow for greater reuse of our components with 3rd party groupware and applications using other implementation technologies.

SUMMARY

Many applications require various kinds of groupware functionality. We have developed a range of plug-in groupware components that provide various group awareness capabilities. The key contributions of our work include:

- Our approach to extending component-based systems by dynamically plugging in appropriate groupware-supporting components. These extend applications to provide the desired groupware capabilities in a seamless fashion. Groupware component facilities can also be statically specified by application developers, and be turned off (unplugged) by end users if not required.
- Our approach is elegant in that no code modifications to neither groupware nor application components are necessary - the architectural facilities of our JViews component framework are leveraged to obtain the necessary application component (and groupware component) reconfiguration to provide the groupware facilities.
- We have developed an architectural model, framework design and implementations that support a wide range of plug-in groupware capabilities, each expressed in discrete, separately reusable software components. These groupware components are mutually compatible and where possible share similar user interface, persistency and distribution components.
- We have developed a flexible event subscription and notification mechanism, and aspect-based introspection and decoupled interaction mechanism, both used to realise out groupware components. Such architectural and framework design features, and our implementation techniques for them, can be used to support other kinds of dynamic component-based systems extension and reconfiguration (such as for persistency and distributed processing, partially explored during our development of groupware components).

Our groupware components are generic and have been successfully reused in a diverse range of applications, providing a wide range of collaborative work-supporting facilities required by end users. Based on our experiences, we suggest the following to others developing component-based groupware applications:

- Carefully identify component responsibilities within the architecture aiming to maximise reuse of abstractions e.g. user interface components, event broadcasting components, data persistency components, locking co-ordination, inter-person messaging, and so on. A good set of component abstractions makes groupware development much easier and results in more maintainable solutions.
- Peer-to-peer architectures work well and provide robust groupware facilities, but require more complex data mirroring, data synchronisation and have a degree of unproven scalability, particularly in the presence of a large number of event exchanges.
- Components need to be engineered to support event-based interaction, ideally before-change and after-change event subscription for synchronous groupware facilities, and a well-defined set of common events makes integrating components easier.
- Components need to publicise information about their events and services in order to allow run-time plug and play and suitable automatic adaptations to component configurations to be built. This is not easy, and requires careful thought, design and implementation. Our aspect characterisations are one of many possible techniques that can be used, though emerging standards for new component technologies like .NET and J2EE may provide more generic support for component introspection.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the many helpful comments of the anonymous reviewers on earlier versions of this paper.

REFERENCES

1. M. Roseman and S. Greenberg, Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction* 3, 1 (March 1996), 1-37.
2. T.C.N. Graham, C.A. Morton, C.A. and T. Urnes, ClockWorks: Visual Programming of Component-Based Software Architecture. *Journal of Visual Languages and Computing*, (July 1996), 175-19
3. T. Urnes and T.C.N. Graham, Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations. In *Proceedings of Design, Specification and Verification of Interactive Systems*, 1999.
4. R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson and W. Wilner, The Rendezvous Architecture and Language for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction* 1, 2 (June 1994), 81-125.
5. R.D. Hill, The Abstraction-Link-View Paradigm: Using Constraints To Connect User Interfaces to Applications. In *Proceedings of CHI '92: Human Factors in Computing*, ACM Press, 1992, pp. 335-342.
6. P. Dewan and R. Choudhary. (1991) Flexible user interface coupling in collaborative systems. In *Proceedings of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.
7. C. Shuckman, L. Kirchner, J. Schummer and J.M. Haake, Designing object-oriented synchronous groupware with COAST. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM Press, November 1996, pp. 21-29.
8. R.I. Furguson, N.F. Parrington, P. Dunne, J.M. Archibald and J.B. Thompson, MetaMOOSE – an Object-oriented Framework for the Construction of CASE Tools. In *Information and Software Technology* 42, 2 (January 2000).
9. M. Roseman and S. Greenberg, Simplifying Component Development in an Integrated Groupware Environment. In *Proceedings of the ACM UIST'97 Conference*, ACM Press, 1997.
10. G.H. ter Hofte and H.J. van der Lugt, CoCoDoc : A framework for collaborative compound document editing based on OpenDoc and CORBA. In *Proceedings of the IFIP/IEEE international conference on open distributed processing and distributed platforms*, Toronto, Canada, May 26-30, 1997. Chapman & Hall, London, 1997, p. 15-33.
11. S.M. Kaplan, W.J. Tolone, A.M. Carroll, D.P. Bogia and C. Bignoli, Supporting Collaborative Software Development with ConversationBuilder. In *Proceedings of the 1992 ACM Symposium on Software Development Environments*, ACM Press, pp. 11-20.
12. Microsoft Corp., *Microsoft NetMeeting 2.1*. See: <http://www.microsoft.com/netmeeting/>.
13. C.A. Ellis, S.J. Gibbs and G.L.Rein, Groupware: Some Issues and Experiences. *Communications of the ACM* 34, 1 (January 1991), 38-58.
14. B. Magnusson, U. Asklund and S. Minör, Fine-grained Revision Control for Collaborative Software Development. In *Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, Los Angeles CA, December 1993, ACM Press, pp. 7-10.
15. S. Bandinelli, E. DiNitto, and A. Fuggetta. Supporting cooperation in the SPADE-1 environment. *IEEE Transactions on Software Engineering* 22, 3 (December 1996), 841-865.
16. I.Z. Ben-Shaul, G.T. Heineman, S.S. Popovich, P.D. Skopp, A.Z. Tong, and G. Valetto, Integrating Groupware and Process Technologies in the Oz Environment. In *Proceedings of the 9th International Software Process Workshop: The Role of Humans in the Process*, IEEE CS Press, Airlie, VA, October 1994, pp. 114-116.
17. J.C. Grundy, J.G. Hosking, W.B. Mugridge and M.D. Apperley, A decentralised architecture for software process modelling and enactment. *IEEE Internet Computing* 2, 5 (Sept/Oct 1998), IEEE CS Press, pp. 53-62.
18. J.C. Grundy and J.G. Hosking, Developing Adaptable User Interfaces for Component-based Systems. *Interacting with Computers* 14, 2 (March 2002), Elsevier Science Publishers, pp 175-194.
19. J.C. Grundy, W.B. Mugridge and J.G. Hosking, Constructing component-based software engineering environments: issues and experiences. *Journal of Information and Software Technology* 42, 2, (January 2000), pp. 117-128.
20. T.C.N. Graham and J.C. Grundy, External Requirements of Groupware Development Tools. In *Engineering for Human-Computer Interaction*, Chatty, S. and Dewan, P. Eds, Kluwer Academic Publishers, August 1999, 363-376.
21. P. Dourish and V. Bellotti, Awareness and coordination in shared workspaces, In *Proceedings of the 1992 Conference on Computer-Supported Cooperative Work*, Toronto, Canada, 1992, ACM Press, pp. 107-113.
22. A.J. Dix, Computer-supported cooperative work – a framework. In *Design Issues in CSCW*, Rosenberg, D. and Hutchison, C., Springer Verlag, 1994, pp. 23-37.
23. R. Bentley, T. Horstmann, K. Sikkell and J. Trevor, Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. In *Proceedings of the 4th International WWW Conference*, Boston, MA, December 1995.
24. L. Wakeman and J. Jowett, *PCTE: the standard for open repositories*. Prentice-Hall, 1993.
25. A.W. Brown and K.C. Wallnau, Current state of Component Based Software Development. *IEEE Software* (Sept/Oct 1998), 37-46.
26. C.A. Szyperski, *Component Software: Beyond Object-oriented Programming*. Addison-Wesley, 1997.
27. D.F. D'Souza and A. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.

28. P. Allen and S. Frost. *Component-Based Development for Enterprise Systems: Apply the Select Perspective*TM, SIGS Books/Cambridge University Press, 1998.
29. R. Sessions, *COM and DCOM: Microsoft's vision for distributed objects*. Wiley, 1998.
30. Apple Computer Inc., *OpenDoc Users Manual*, 1995.
31. S.M. Kaplan, G. Fitzpatrick, T. Mansfield and W.J. Tolone, Shooting into Orbit. In *Proceedings of Oz-CSCW'96*, University of Queensland, Brisbane, Australia, August 1996.
32. G.E. Kaiser and S. Dossick, Workgroup middleware for distributed projects, In *Proceedings of IEEE WETICE'98*, Stanford, June 17-19 1998, IEEE CS Press, pp. 63-68.
33. W. Emmerich, CORBA and ODBMSs in Viewpoint Development Environment Architectures. In *Proceedings of the 4th International Conference on Object-Oriented Information Systems*, Springer Verlag, 1997, pp. 347-360.
34. M. Mezini and K. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of OOPSLA '98*, Vancouver, WA (October 1998), ACM Press, pp. 97-116.
35. M. Mezini, L. Seiter and K. Lieberherr, Component Integration with Pluggable Composite Adapters. *Software Architectures and Component Technology*, M. Aksit, Ed, Kluwer, 2000.
36. J.L. Pryor and N.A. Bastan, Java Meta-level Architecture for the Dynamic Handling of Aspects. In *Proceedings of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas (June 26-29 2000), CSREA Press.
37. J.C. Grundy, Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering* 10, 6 (December 2000).
38. R. Monson-Haefel, *Enterprise JavaBeans*. O'Reilly, 1999.
39. J. O'Neil and H. Schildt, *Java Beans Programming from the Ground Up*. Osborne McGraw-Hill, 1998.
40. G.E. Krasner and S.T. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1, 3 (1988), 8-22.
41. J.C. Grundy, J.G. Hosking and W.B. Mugridge, Supporting flexible consistency management via discrete change description propagation. *Software - Practice and Experience* 26, 9 (September 1996), Wiley, 1053-1083.
42. R.B. Dannenberg, A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors. *Software-Practice and Experience* 20, 2 (February 1990), 109-132.
43. M.R. Wilk, Change Propagation in Object Dependency Graphs. In *Proceedings of TOOLS US '91*, Prentice-Hall, August 1991, pp. 233-247.
44. J.C. Grundy, Engineering Component-based, User-configurable Collaborative Editing Systems. In *Proceedings of 1998 International Conference on Engineering for Human-Computer Interaction*, Crete, Greece, Sept 14-18 1998, Kluwer Academic Publishers.

6.3 A generic approach to supporting diagram differencing and merging for collaborative design

Mehra, A., Grundy, J.C. and Hosking, J.G. A generic approach to supporting diagram differencing and merging for collaborative design, In *Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering*, Long Beach, California, Nov 7-11 2005, IEEE Press, pp. 204-213

DOI: [10.1145/1101908.1101940](https://doi.org/10.1145/1101908.1101940)

Abstract: Differentiation tools enable team members to compare two or more text files, e.g. code or documentation, after change. Although a number of general-purpose differentiation tools exist for comparing text documents very few tools exist for comparing diagrams. We describe a new approach for realising visual differentiation in CASE tools via a set of plug-in components. We have added diagram version control, visual differentiation and merging support as component-based plug-ins to the Pounamu meta-CASE tool. The approach is generic across a wide variety of diagram types and has also been deployed with an Eclipse diagramming plug-in. We describe our approach's architecture, key design and implementation issues, illustrate feasibility of our approach via implementation of it as plug-in components and evaluate its effectiveness.

My contribution: Co-developed key ideas for this research, co-designed approach, co-supervised Masters student, wrote substantial parts of paper, co-lead investigator for funding for this project from FRST

A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design

Akhil Mehra
Dept. Computer Science
University of Auckland
Auckland, New Zealand
+64-9-3737-599

akhilmehra@gmail.com

John Grundy
Dept. Electrical and Computer Eng.
University of Auckland
Auckland, New Zealand
+64-9-3737-599 ext 88761

john-g@cs.auckland.ac.nz

John Hosking
Dept. Computer Science
University of Auckland
Auckland, New Zealand
+64-9-3737-599

john@cs.auckland.ac.nz

ABSTRACT

Differentiation tools enable team members to compare two or more text files, e.g. code or documentation, after change. Although a number of general-purpose differentiation tools exist for comparing text documents very few tools exist for comparing diagrams. We describe a new approach for realising visual differentiation in CASE tools via a set of plug-in components. We have added diagram version control, visual differentiation and merging support as component-based plug-ins to the Pounamu meta-CASE tool. The approach is generic across a wide variety of diagram types and has also been deployed with an Eclipse diagramming plug-in. We describe our approach's architecture, key design and implementation issues, illustrate feasibility of our approach via implementation of it as plug-in components and evaluate its effectiveness.

Categories and Subject Descriptors

D.2.2 [[Software Engineering]] Design Tools and Techniques – CASE tools

H.5.3 [Information Systems] Group and Organization Interfaces – asynchronous interaction

D.2.7 [Software Engineering] Distribution, Maintenance, and Enhancement – version control

General Terms

Design, Human Factors

Keywords

visual differencing, merging, version control, CASE tools.

1. INTRODUCTION

Efficient management of software artefacts is a major task of any project involving more than one person. An important goal is

ACM COPYRIGHT NOTICE. Copyright © 2005 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

maintaining versions of software artefacts as they evolve, preventing people from accidentally overwriting each other's work, and allowing tracking of changes made to those artefacts over time [1], [3], [21].

A related and important task is support for version comparison and merging [1], [21]. Although configuration management tools provide good support for versioning, tool support is also needed to identify differences between versions of an artefact. The Unix tool diff [11] is one such popular tool for comparing two text files. Diff compares files and indicates a set of additions and deletions. Many version control tools provide functions similar to diff to identify changes between versions of text documents. Initially diff tools were hard to use as users needed to manually navigate to lines where changes were made. This led to creation of visual differentiation tools, to improve usability and highlight changes within IDEs. Many of these are generic working across many text-based document types [10], [15]. Many IDEs incorporate such differentiation facilities, often using a diff-style tool as a component-based plug-in to do the comparison with results presented visually in the IDE, as in the Eclipse version tree plug-in [2]. Merge support in an IDE uses differences detected to apply changes made in one version of a document to another.

Although good, generic support is available for differentiating and merging text documents, limited support is currently available for differentiating graphical objects such as UML design diagrams and software architectures [16], [22]. Providing visual differentiation in CASE tools is important to enhance a team's efficiency and effectiveness when collaborating asynchronously using diagrams to represent information [20]. Existing diagram differentiation tools are usually limited to a single diagram type and hard-coded into the CASE tool. A diff-style algorithm doesn't work for two (or three) dimensional diagrams as the isolation of the diagram into "islands of difference" is very difficult. Tool developers use diagram type-centric techniques that need recoding for different diagram types.

We have added generic visual differentiation facilities to Pounamu [23], a meta-CASE tool which allows a user to specify and generate multi-view visual design tools. Pounamu diagrams are stored in XML format so visual differentiation facilities were added by differencing this XML format. Differences are then translated into editing events which are presented to users by using appropriate highlighting to emphasize differences. Users also have the ability accept/reject changes made thus enabling partial merging of diagrams.

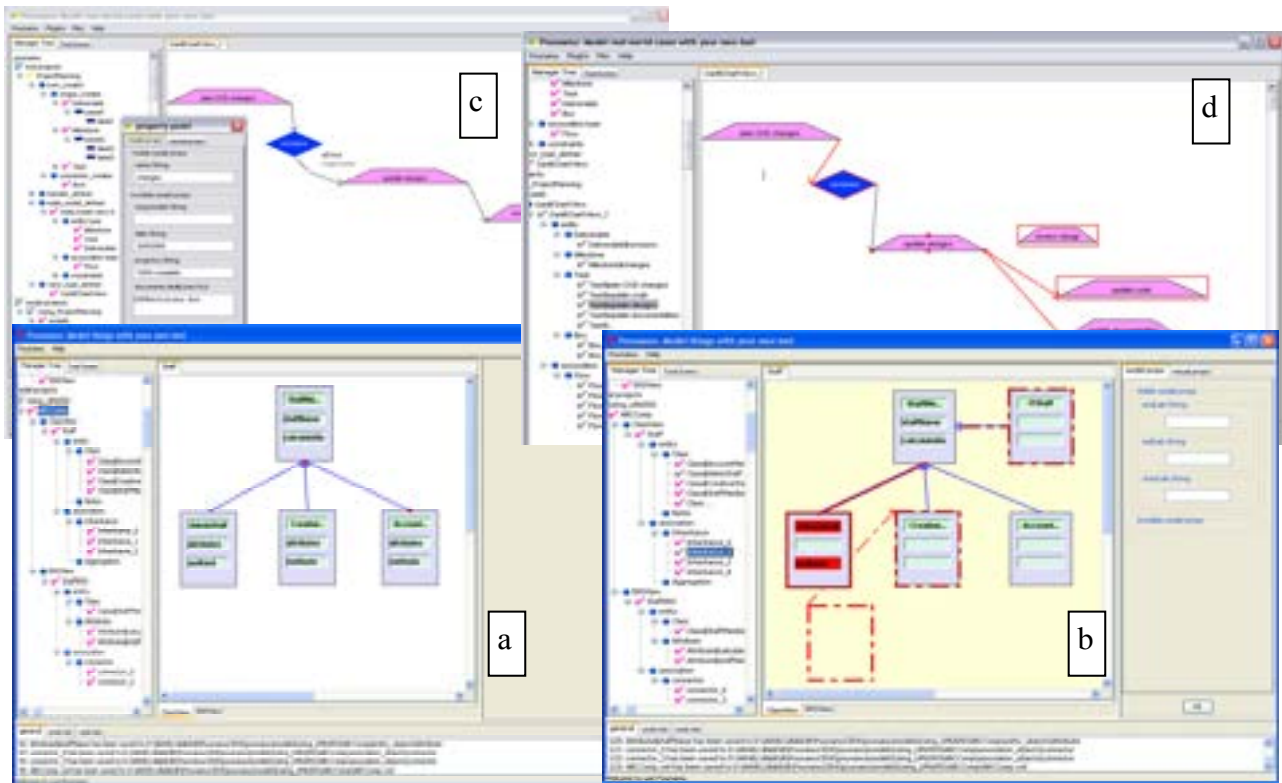


Figure 1. Examples of editing tools in Pounamu.

Syntactic and some semantic conflicts are detected and presented. Use of our technique on a range of diagram types has proven successful, and usability evaluations demonstrate its effectiveness.

We first present motivation for our work and related research. We then present our approach for providing versioning, visual differentiation and merging support in Pounamu and illustrate user interaction with our plug-ins. An Eclipse graphical editor plug-in for Pounamu-specified diagrams using the same differencing and merging plug-ins is also shown. We then describe key design and implementation decisions, evaluate our approach, and identify directions for further research.

2. MOTIVATION

Consider two users modifying the design of a system. These developers may work together to review and modify the design, but at times may be working independently and need to make design modifications. Figure 1 (a) shows part of a UML design for a software system that may be under modification by user “John”. John may add classes, attributes, operations and relationships. He may modify existing classes and relationships. He may also reposition diagram elements, change relationship role names and arities, and possibly annotate the design diagram with notes or other supported annotations. User “Mark” will want to be isolated from these changes for a time, requiring versioning of design diagrams. Eventually Mark will need to compare the version of the design he has to that of John’s via a diagram differencing facility, highlighting changes as shown in Figure 1 (b). Figure 1 (c) and (d) show other examples of diagram difference presentation, in this case a Gantt chart. Mark may want

to merge some or all of John’s changes into his version, which may include some of Mark’s own independent modifications. Other kinds of diagrams John and Mark are asynchronously editing will also need to be versioned and differentiated and merged.

The diagrams in these examples are implemented using the Pounamu meta-tool [23]. Pounamu is a meta-tool for building domain-specific visual language diagramming tools. Pounamu also provides a framework for realizing and using the specified tools. Using Pounamu a user can rapidly specify visual notational elements, underlying tool information model requirements, visual editors, the relationship between notational and model elements, and behavioral components [23]. Tools are generated dynamically and can be used for modeling immediately.

Efficient asynchronous collaboration amongst groups of people working together may be facilitated using versioning and differentiation tools. We were motivated to improve asynchronous collaboration in Pounamu by providing users the ability to version Pounamu model projects via a CVS repository [9]. The ability to version Pounamu model projects led immediately to the requirement for generic visual differentiation and merging tools for diagram versions.. Such a tool should enable users to visually compare their current work with prior versions or other user’s versions of the same diagram. It should also provide users with information about addition, deletion, movement or property changes in shapes or associations and users must be able to specify changes they wish to keep and changes they wish to discard thus enabling them to merge their current diagram with any prior version. Syntactic conflicts should also be presented to users for resolution, and any semantic conflicts

introduced by a merge should be highlighted. As Pounamu supports building a huge range of domain-specific diagramming tools, the visual differentiation and merge facilities should be generic across any diagram type. Finally, we wanted to seamlessly add version control, diagram differentiation and merging to Pounamu using its plug-in API, rather than modifying its code directly.

Much research has been done on the issue of version control, differencing and merging support for programming language editors and other (textual) document editors. Various version control tools have been developed and made available as remote services and plug-ins for many IDEs, such as CVS [7] and RCS [21]. More complex versioning facilities are supported in some specialized program editors, such as Mjølner [14], making use of abstract syntax graphs to link fine-grained versions of artifacts.

Many textual differencing tools share the approach of diff [11] and related tools, determining a set of additions and deletions to change file A into file B [10], [15]. However, extensions to this have been made to support binary and, more recently, XML file differencing, such as the IBM and Stylus Studio XML Diff/Merge tools [12], [19]. Several IDEs with diagramming tools provide model-based differencing, some using custom approaches and others XML-based model differencing [17] [16], [18]. The approach of Ohst et al [17] use a model of drawings and version histories to detect changes and present to users via diagram colour annotation. Several tools [16], [18] use XML differencing of the model structures to detect changes and support graphical annotation of the XML to indicate changes between compared versions. They provide interactive user acceptance or rejection of changes. While comparing design diagrams at the model level has advantages of reuse of diff-style differentiation and merging tools, presentation of the differences textually does not give a very satisfactory sense of actual diagram comparison to the user.

Work has been done providing custom differencing algorithms in software tools, such as architecture design environments. These however tend to use customized algorithms specific to one graphical model type rather than a general approach [25], [22], [17]. Generally there is poor support for differentiating graphical objects such as UML diagrams at the visual, diagrammatic level. Two exceptions are IBM Rational Rose [13] and Magic Draw UML 9.0 [16]. Both tools convert their diagrams into hierarchical text and then perform differentiation on this hierarchy. Changes are shown using highlighting schemes on the text. The main drawback of this approach is that changes are no longer visible in graphical form and thus more difficult to comprehend. The only tools that we are aware of that present changes graphically are a prototype UML editor built at GroupLab [20], and a UML diagram differencing tool we developed in earlier work that leveraged change event objects passed between users' CASE tools to form a delta capturing version differences [5]. This highlighted changes between diagrams by annotation, presenting version differences graphically.

3. OUR APPROACH

Our aim in this work was to extend our Pounamu meta-tool to better support asynchronous collaborative work with diagrams. Key requirements were supporting versioning, differentiation and merging across any diagram type using a set of plug-in extensions. We wanted to present diagram differences graphically and allow users to interactively select differences to accept

between versions. Figure 2 shows our approach to developing this generic Pounamu diagram version, differentiation and merging support.

Users create Pounamu diagrams and may check these into a remote CVS server. This is facilitated via a plug-in added to Pounamu to support check-in/check-out to a CVS server. Another user may check out a Pounamu diagram from a CVS repository (1). If another user currently has a copy of the diagram, an alternate version is created enabling both to modify it. This new version of the diagram is then modified by the developer using Pounamu's graphical editing tools (2). The developer may then make the modified version accessible to others by checking it back into the CVS repository (3). Another user, for example the original diagram creator, may check this alternate version of the diagram out (4) and then apply the differencing plug-in to compare changes between the two versions of the diagram. Our Pounamu diagram differencing plug-in decomposes the XML describing a model project into a Java object graph. These Java objects are then compared to identify differences. The differences are translated into Pounamu Command objects, each of which embodies a set of API calls that describe editing events that take place in a Pounamu model project (e.g. add shape, connect shapes, move shape, set shape property, delete connector etc) (5). These generated Pounamu Command objects represent the set of changes that need to be made to one diagram version to convert it into the other.

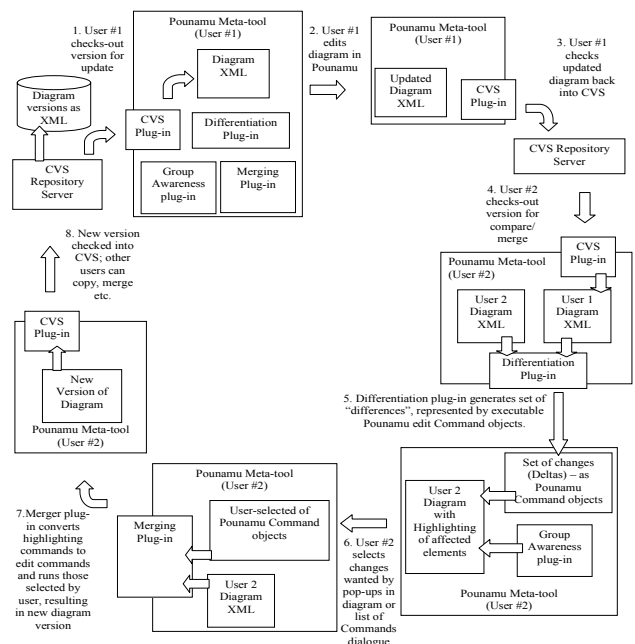


Figure 2. Our Approach to supporting generic diagram versioning, differentiation and merging in Pounamu.

In earlier work we have developed group awareness facilities for Pounamu as part of the research carried out to add plug-in collaborative editing support [9]. These facilities enable us to provide awareness information to users while they are collaboratively editing diagrams by highlighting other users' additions, modifications and deletions to a diagram in near-real time. As part of creating the group awareness component for Pounamu we developed a core set of highlighting schemes and an

API to the plug-in for depicting changes in a visual diagram. One of our goals while developing the visual differencing and merging plug-ins for Pounamu was to reuse the highlighting support provided by our group awareness plug-in to graphically decorate one version to highlight the differences between the diagrams for the user (6). The user is then able to see in the diagram view in Pounamu differences between model project versions and can interactively select the changes to accept or reject (7). Accepting a change causes its associated Pounamu Command to be executed, updating the diagram and thus providing selective merging of diagrams. The merged diagram can be checked back into the CVS repository for others to use (8).

4. ARCHITECTURE

Each Pounamu model project consists of a set of model entities and associations. A number of diagrams, or views, are provided of the model entities and associations allowing users to view and edit the model information. Each view contains a number of shapes and connectors. Shapes are linked by connectors (with owning parent and child shape). Connectors may be visible e.g. lines between shapes, or invisible e.g. representing containment of a set of shapes by another, layout constraints, etc. Every connector or shape has a number of attributes. Each shape has a unique persistent identifier, an objectID, and also a “rootID”, which is the objectID of the root version of a shape object. Shapes in different versions can be identified uniquely by their objectID and different versions of a shape derived from the same root are identified by sharing the same rootID value.

We represent Pounamu model elements and view elements as XML files for storage, as shown in Figure 3. Our experience developing and evaluating various collaborative model-view based diagramming tools has shown that model differencing alone is insufficient to effectively support diagram editing, differencing and merging [5], [6], [9]. We need to difference diagram-level data and by merging detected changes, the underlying model is also updated.

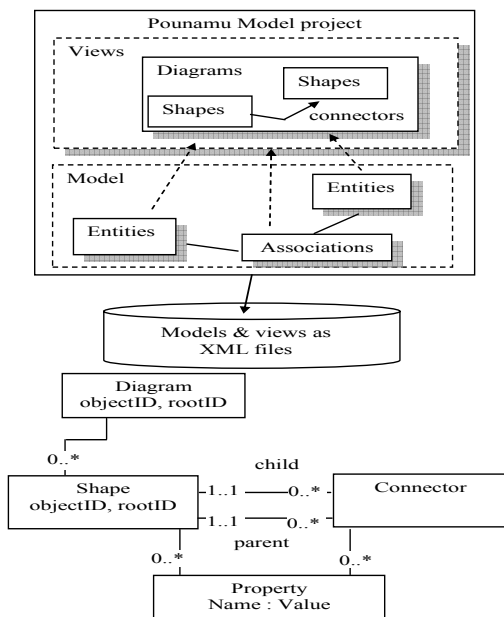


Figure 3. Pounamu model projects, views and the generic Pounamu diagram data structure.

Figure 4 illustrates the key architectural components and their interactions in our extended Pounamu environment. A CVS client plug-in is used to check-out a diagram version (1), which is then converted from its XML save format into Java objects (2) and then rendered and displayed to the user. Edits to the diagram (3) update the internal Java object structure. When comparing diagram versions, an alternative version of the diagram is retrieved (4) and the two versions compared by our diagram differentiation plug-in (5). This generates a set of Pounamu Command objects to describe the changes (delta) between the two diagram versions (6). The diagram highlighting plug-in from our synchronous collaborative editing system for Pounamu is used to highlight changes in the diagram (7). A diagram merging plug-in uses the Command objects to update one diagram version (8), generating a new, merged version for check-in to the CVS repository (9).

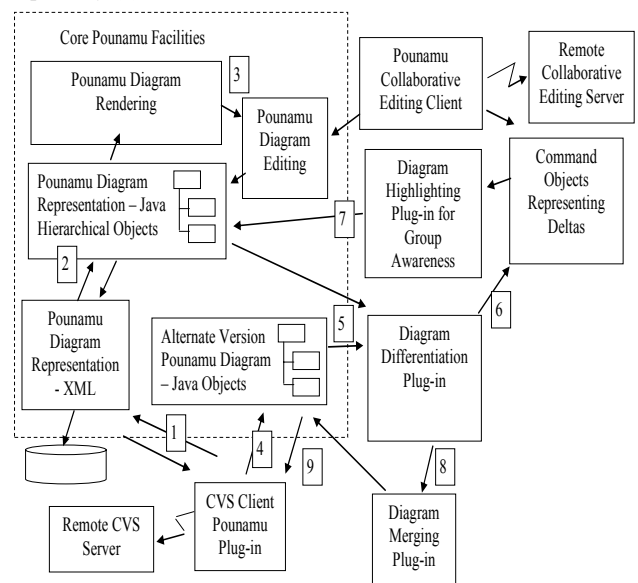


Figure 4. Architecture of the extended Pounamu Environment.

Figure 5 presents pseudo code describing our view differencing algorithm. Differences between views are determined by iterating through all views in a Pounamu model project and comparing each view with corresponding views in another version of the model project. We generate a list of editing Command objects that, when run on one version, convert it to the other version. The set of Command objects generated correspond to line inserts, updates and deletes in a CVS-style textual differencing algorithm.

Let us assume we are comparing two different versions of a certain Pounamu view, view1 and view2 respectively. We compare these two alternative versions of the same view on the basis of shapes and connectors present in each view. We build up a set of Pounamu Command objects that encapsulate the changes that would need to be made to view2 to convert it to view1. The two operations, diffShapes() and diffConnectors(), are used to identify the differences between two views. The operation highlightChanges() is used to iterate over the change list (made up of generated Command objects) and highlight one version to indicate differences to the user.

```

/* Differentiate two different views*/
diffViews(view from project 1, view from project 2) {
  diffShapes(view1, view2, changesList)
  diffConnectors(view1, view2, changesList)
  highlightChanges(view2, changesList)
}

/* Differentiate relating to shapes in a particular view.*/
diffShapes(view1, view2, changesList) {
  Vector view1PounamuShapes = view1.getShapesVector();
  for all (existingShape : shapes in View1) {
    /* Get shape in second view with the same shape root id */
    PounamuShape secondViewShape = view2.findShape(existingShape.getRootID());
    if (secondViewShape != null) { /* if same-rooted shape exists... */
      /* Check if all attributes match. If same do nothing; else find differences */
      if (do the two shapes have the same position) {
        /* If false add to shapes that have been moved */
        changesList.add(MoveShapeCommand(secondViewShape, dx, dy));
      } /* Check if shape has been Resized */
      if (are the two shape of the same size) {
        /* If false add to shapes that have been resized */
        changesList.add(ResizeShapeCommand(secondViewShape, dwidth, dheight));
      } /*Attributes Changed */
      if (check if shape attribute values are the same) {
        for all ( prop : props of existingShape where oldValue=existingShape.getValue(prop) !=
              newValue=secondViewShape.getValue(prop)
            ) {
          changesList.add(ChangePropertyCommand(secondViewShape, prop, oldValue, newValue));
        }
      }
    } else {
      /* shape does not exist in view2 so it has been deleted... */
      changesList.add(DeleteShapeCommand(secondViewShape));
    }
  }
  Vector view2PounamuShapes = view2.getShapesVector();
  for all ( secondViewShape : shapes in View2 ) {
    /* Get shape in other view with the same shape root id */
    PounamuShape existingShape = view1.findShape(secondViewShape.getRootID());
    if ( existingShape == null ) {
      /* shape does not exist in view1 so have added it */
      changesList.add(NewShapeCommand(view1, secondViewShape.getType(), secondViewShape.getRootID()));
      for all ( prop : properties of secondViewShape )
        changesList.add(ChangePropertyCommand(newShape, prop, null, secondViewShape.getValue(prop)));
    }
  }
}

/* highlight changes in view */
highlightChanges(view2, changesList) {
  for all ( change : changes in changesList) {
    if ( change == NewShapeCommand )
      shape = NewShapeEvent.getShape();
      shape.setTempProperty("oldLineColor", shape.getLineColor()); // remember old values...
      ...
      shape.setLineColor(red);
      shape.setBorderThickness(2);
      ...
    else if ( change == MoveShapeCommand )
      ...
  }
}

```

Figure 5. Pseudo Code Describing the Model Project Differ.

As views are made up of shapes and connectors they are compared on this basis. All Pounamu views, shapes and connectors have a unique, persistent object ID (a GUID) generated when they are created, making them uniquely identifiable across users' model projects. View elements are similarly tagged with a unique object "root ID" in addition to their own unique object ID. If a view is created from an existing view i.e. an alternative version or revision, the new view's elements are tagged with the root ID of the elements they are derived from. If a view is created from scratch, each element's root ID is the same as its object ID.

When comparing views the root ID is used to identify view elements with the same common root object i.e. elements in each view that are alternative versions of each other. This can be thought of as a simplified form of origin analysis [24] and is a way of implementing object uniqueness [17]. Connector alternative versions are identified by their source/target shape root IDs. We use a two-pass approach over the diagram's shape and connector elements rather than graph-based traversal [25] to generate the Command objects representing a delta between them. Firstly we iterate over each shape in view1 and see if it exists in view2. If not, the shape must be deleted to convert view1 to view2 (which will result in all of its connectors also being

deleted). A delete shape Command object is generated to represent this action for each shape in view1 but not present in view2. Shapes present in both view1 and view2 have their size, location and other properties compared. Editing commands are generated to modify any non-matching properties. We then look for shapes in view2 not present in view1. Any found must be added to view1 to convert it to view2, so shape addition Commands are constructed and shape initial size, location and property value setting commands are generated to initialize the newly added shape. We then pass over the connectors using a similar approach: locate connectors present in both views and generate property change Commands to synchronise their properties; generate connector delete Commands to remove connectors in view1 not in view 2; and generate connector Create and property initialization Commands to add connectors that are in view2 but not in view1. We have found this two-pass approach to be sufficient for Pounamu views and generally less complex to implement and more efficient on large views than graph traversal approaches.

The diffShapes() method iterates through all shapes in a view and tries to find shapes with the same root ID in the view that it is being compared to. If the shape can't be found, it either needs to be deleted or added to synchronize the two views. If the shape

exists then it checks for differences in a shape's size, location and attribute values. All differences are recorded as Pounamu Command objects (NewShapeCommand, ResizeShape Command ChangePropertyCommand etc). These Pounamu Command objects may be executed in order to synchronise the two views which results in view1's diagram data structure being converted into the same as view2's. Any newly added shapes in view1 have their rootID set to the rootID of their view2 shape they have been derived from, allowing identification of common ancestors for shapes. Complex shape structures e.g. containment and layout-based constraints are supported by our comparison and merging mechanism as they are driven by change events generated by the Command object execution.

The Diagram Highlighting plug-in from our collaborative diagram editing client provides a highlightChanges() function. This decorates the graphical diagram rendering to indicate changes required to convert one version to the other. The diagram highlighter iterates through the generated Pounamu Command objects and for each modifies the Pounamu diagram elements – bold red for added shapes; dashed fine line around deleted; red fill box for changed shape and connector properties; and dashed origin and line for move/resize of shapes. The highlighter modifies standard properties of shapes and connectors to achieve some of these highlights and adds its own annotation shapes and connectors to the diagram to achieve others. Upon carrying out merging of changes (i.e. execution of editing Commands), standard Pounamu diagram editing event propagation notifies the highlighting component which then removes annotations.

Users may select a subset of Commands to apply to a view to effect merging of changes by interacting with the decorated diagram elements. Accepting a change and having its associated Command object executed results in a change-by-change partial merging. This may result in some commands not being able to be applied. For example, a NewShapeCommand is not applied meaning a subsequent ChangePropertyCommand on the shape not being able to be applied to the view. Similarly during merging semantic errors can be produced e.g. two classes with the same name or an invalid type association between objects. We allow the Pounamu tool's semantic constraint handling to detect and highlight these as merging proceeds. However, applying the Command objects (i.e. modifying the syntactic structure of the view) and providing the user a list of semantic errors introduced into the new version could be supported in advance of user-acceptance of syntactic merge changes.

5. EXAMPLE USAGE

We illustrate our diagram differentiation and merging algorithm with a simple class diagramming tool example. Two colleagues, John and Tim, are working with a Pounamu-created UML design tool at different locations. By using Pounamu versioning capabilities they can asynchronously collaborate on a Pounamu model project. In order to find differences between different versions of model project's views they are provided with our CVS version control plug-in, diagram differencing plug-in and difference merging plug-in.

A CVS repository enables storing and versioning of binary and text files and sharing of these files among distributed users. Asynchronous collaboration in Pounamu is facilitated by versioning Pounamu model projects and their views using a shared CVS repository via a plug-in to the Pounamu modeling

tool. This CVS plug-in enables storing and retrieving Pounamu model project information and individual diagram information in their XML save file format from a shared, remote CVS repository. Users who wish to collaborate on a model project asynchronously are expected to check in (store) their model projects into a CVS repository. Remote collaborators can check out (retrieve) model projects thus enabling asynchronous collaboration.

Let us assume John creates a new UML class diagram for an existing Pounamu UML tool model project that he and John are working on. John adds shapes and connectors, moves and resizes shapes, sets various properties and so on to create his class design diagram. In order to share these changes with Tim, John "checks in" his model project with the CVS repository. The current state of the new class diagram checked in by John is illustrated in Figure 6.

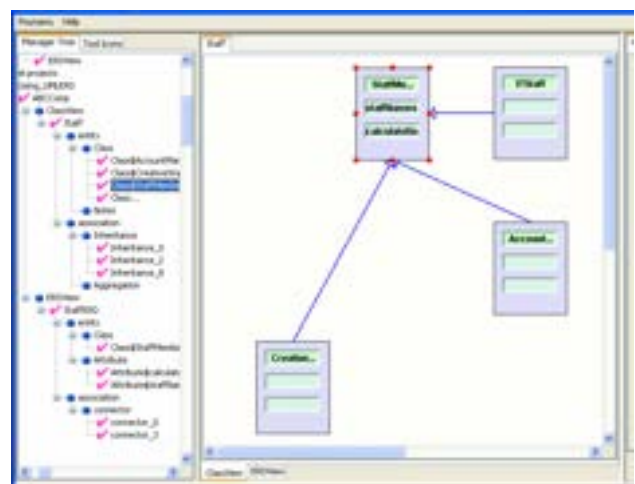


Figure 6. Initial class diagram view as checked in by John.

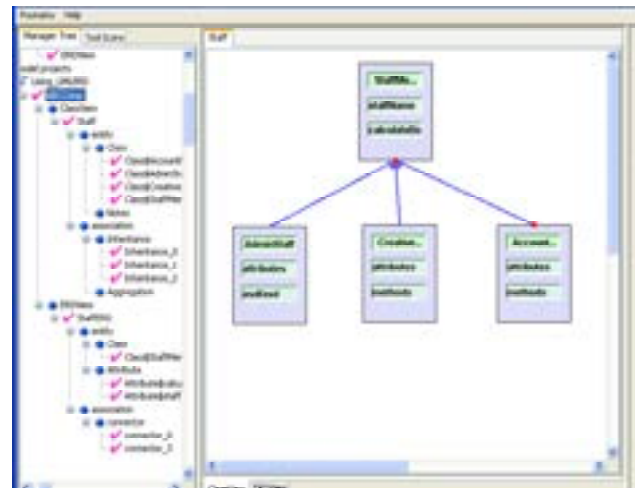


Figure 7. Tim's alternative version of the class diagram.

Tim subsequently checks out the model project containing this same UML class diagram from the CVS repository, thereby creating an alternative version of it and asynchronously makes changes to his version of the class diagram. Tim's alternative version of the diagram after making several changes is shown in Figure 7. Class icon shapes have been moved, deleted and some

of their properties changed; association connectors have been added and deleted. Tim checks his model project back into the CVS repository, resulting in his alternative versions of changed diagrams being checked in as well.

In order to see any changes that might have been made by Tim, John differentiates his model project against what Tim has checked in. Appropriate pop-up menu items in the right-hand Pounamu tree viewer are provided when the CVS versioning and diagram differentiation plug-ins have been enabled. On selecting the “Diff with Later Versions” menu item for an open diagram view, John is presented with a differentiation dialogue box as shown in Figure 8. John chooses “Version 1.5” checked in by Tim to differentiate his current version (“Version 1.4”) of the class diagram against. On executing the differentiate command Tim’s version is checked out of the CVS repository in read-only mode, and the differentiation algorithm from Figure 5 is applied comparing Tim’s version (view 1) with John’s version (view 2) of the diagram. A set of Pounamu Command objects are generated representing the delta between Tim’s alternative version and John’s version of the class diagram. John’s diagram is then annotated to show the differences by our diagram highlighting plug-in iterating over the generated Command objects.



Figure 8. Differentiate Dialogue Box.

Figure 9 shows the highlighted differences between Tim and John’s versions of the class diagram. Solid lines denote creation of a shape or an association (e.g. “AdminStaff” and its association to “StaffMember”). Dotted lines denote deletion of a shape or association (e.g. “ITStaff” and its association to “StaffMember”). Shape movement is denoted by an empty dotted box pointing to the new place where the shape has moved (e.g. “CreativeMember” repositioning). The background colour of the view has changed to a darker shading to enable users to be able to view the extent of changes to a view at a glance. A white background denotes no change while a very rich pink background

denotes a large number of changes. Dark highlighting of entities and attributes denote creation and modification of values. A list of all changes is also available via a dialogue.

A user is free to accept or reject any changes presented. Pop-up menu items are provided with each change to enable John to accept or reject each of the changes made by Tim as he requires. After careful analysis John has decided to accept the shape and association created while rejecting the shape and association deletion. The resulting view that John sees after accepting and rejecting all change is shown in Figure 10. However note that some changes are dependent on previous changes made e.g. accepting the setting of AdminStaff properties and creation of its association to the StaffMember class requires firstly accepting the creation of the new AdminStaff class icon. If this creation has not been accepted, the subsequent changes are marked as “not able to apply”.



Figure 9. View after Differentiation and highlighting.

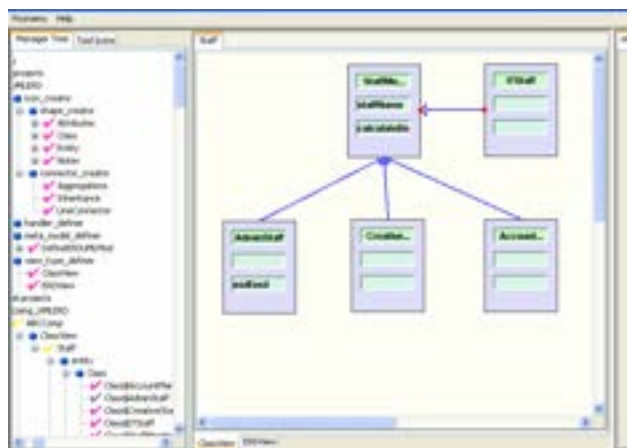


Figure 10. Merged document to be checked in by John.

6. DESIGN AND IMPLEMENTATION

One of the major design goals was to make no changes to the existing single-user Pounamu code when adding our CVS, diagram differentiation and diagram merging capabilities to

Pounamu. Our plug-ins were designed to use a Service Oriented Architecture [9] where each service, collaborative editing, group awareness, and version control, are discovered at run-time by Pounamu environment client plug-ins. The diagram differentiation plug-in was the major development. Design of the visual differentiation tool was made easier by translating differences detected between diagram versions into Pounamu editing Command objects. We then reused the diagram highlighting plug-in from our group awareness support plug-in to highlight a view using these command objects. Version merging was supported by allowing the user to selectively run all or some of these Command objects on the target view, producing a merged view version.

Figure 11 shows the design and interactions of our versioning, differentiating and merging plug-ins. Assume that user John decides to differentiate his existing model project with a later version of Tim's stored in the CVS repository (1). On retrieving Tim's checked in model project as an XML document (2) and having its views loaded into Pounamu's object structures (3), John may differentiate the two (4) using the Differentiation plug-in. A set of Pounamu Command objects (5) are generated by the Differentiation plug-in's diffViews() method, by traversing the view structure of view 1 and comparing each shape and connector to their equivalents (if they exist) in view 2.

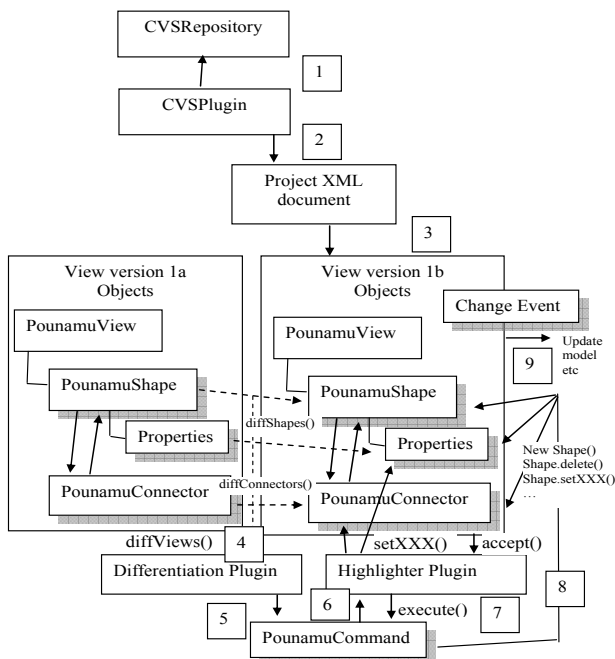


Figure 11. Design of diffing/merge plug-ins.

The sequence of events that take place within diffShapes() and diffConnectors() are quite similar. When diffShapes() is called it retrieves a list of PounamuShape objects in the form of a vector from the first PounamuView (view 1). It then uses this list for comparison. Iterating through the list of PounamuShapes it obtains the rootID for each shape. A rootID uniquely identifies each shape. The rootID is used to retrieve similar objects for the previous version. If the object is not found a NewShapeCommand and a number of SetPropertyCommands are added to the changes list, denoting the need to add this shape if

the two versions are to be synchronised. Similar comparisons are carried out for removal, movement, resizing and changes in shape or connector object properties. The changes list containing the set of PounamuCommands needed to fully convert view 1 to view 2 is then passed to the Highlighting plug-in from our group awareness system (6).

The Highlighting plug-in examines each Command object and sets various visual properties (colour, line thickness and style, border, shadow position and arrow etc) of view 2's Pounamu view objects. The view is re-rendered once this is complete, decorating the view to indicate the version differences. A menu item for each Command object is added to each Pounamu view object's pop-up menu, allowing the user to selectively accept or reject the difference. Accepting the difference tells the Highlighter to run the Command, by calling its execute() method. For each kind of Command, various changes are made to the view e.g. add new shape, delete shape, set shape property etc (8). The Pounamu model is updated when the view objects are thus changed (9), updating any other views sharing the model information.

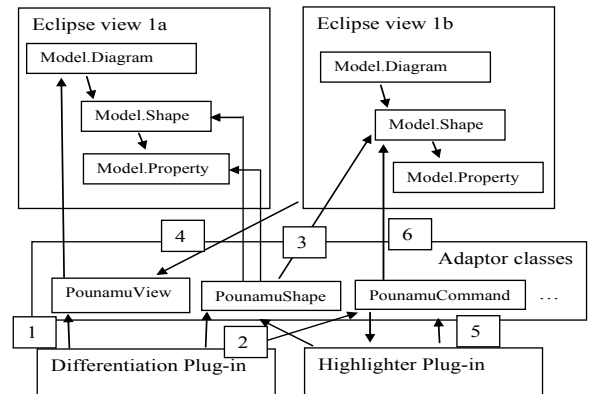
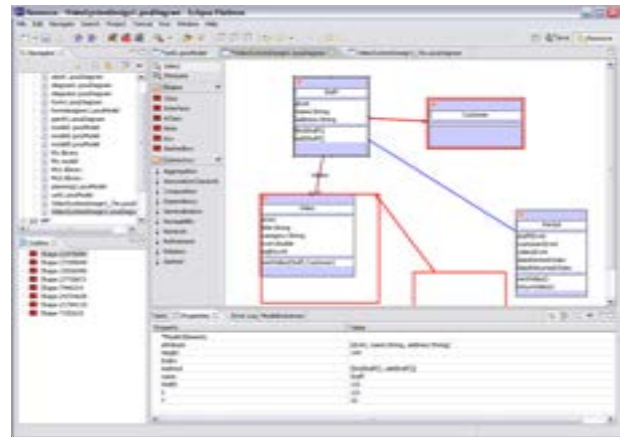


Figure 12. (a) Eclipse Pounamu plug-in differentiation and (b) adaptor objects for the differentiation plug-ins.

We have recently developed an Eclipse plug-in for Pounamu that loads Pounamu meta-tool specifications and provides an Eclipse Graphical Editor Framework (GEF)-based modeling tool. An example of view differentiation with our Differentiation and Highlighter plug-ins is shown in Figure 12 (a). As the Eclipse plug-in does not represent view objects nor Command objects the same way as our Pounamu modeling tool we developed a set of

Adaptor classes to map the methods of our Eclipse plug-in onto compatible named and typed classes of our Pounamu modeling tool. This allowed us to add our Differentiation and Highlighting plug-ins into the Eclipse-based Pounamu environment and support these activities. Figure 12 (b) illustrates the Adaptor objects needed to achieve this. Our Eclipse plug-in does not store the views as XML, hence the need for adaptation to the plug-in's internal object structure.

7. EVALUATION

We used a combination of the Cognitive Dimensions framework [4], Gutwin's groupware assessment framework [8], and a user survey to assess the effectiveness of our Pounamu plug-ins for asynchronous diagram version control, differentiation and merging. The Cognitive Dimensions framework provides a generic approach to measuring various usability characteristics of notations and their environments [4]. We were particularly interested in assessing our plug-ins' support for the dimensions of *visibility of changes*, *hidden dependencies*, *viscosity* (i.e. ease of change of content), *error-proneness*, *hard mental operations*, *consistency* and *closeness of mapping* (i.e. how close a representation matches a user's mental model). From our analysis, key results were that a major benefit of our approach is that differences are highly visible, being presented as graphical annotations to one of the compared diagram versions. Unlike approaches using comparison of models or textual versions of diagrams there are no hidden dependencies between the differences presented and accepted by the user and the resulting merge operation. Interactive acceptance of changes in the diagram by the user reduces both error-proneness and hard mental operations during merging of versions. Our approach to presenting changes is consistent, using colour (red) to denote change, though the difference between some change types is minimal e.g. set property vs create vs move shape. The graphical mark-up of changes in a diagram our approach adopts is similar to the way textual and XML differencing tools [2], [19] present changes providing an element of consistency.

Gutwin's framework [8] provides a groupware-specific set of usability and assessment criteria. As our work is an asynchronous groupware extension to Pounamu, we evaluated our differentiation and merging plug-ins using the *presence*, *authorship*, *identity*, *gaze*, *action*, *intention* and *location* criteria of the framework. Key results from this analysis are that users can identify other's presence and authorship of changes as these are annotated with the author's name and represented in the dialogue view of changes (with changes by multiple authors represented differently in the graphical presentation through tool tips). Changes to diagram content are explicitly and clearly represented and user interaction is explicitly with a graphical representation of changes via pop-ups to accept/reject them. Partial support for intention awareness is supported as the differentiated and visualized changes represent another's (intended) actions for the merged diagram.

We conducted a formal user survey of our group awareness plug-ins for collaborative editing [9]. This involved 10 users carrying out a combination of synchronous and asynchronous design tasks with a Pounamu UML tool over multiple sessions. These were mainly UML diagram review, creation, editing and discussion. Users performed various asynchronous UML diagram editing tasks with our versioning, differentiation and merging plug-ins.

They also performed synchronous UML diagram editing tasks with other Pounamu groupware plug-ins. Feedback on our asynchronous editing support features was very positive, including their response time, approach to presenting changes, support for incremental change accept/reject and overall support for asynchronous diagram-based design activities. Some users requested control over the way changes are presented by the highlighting plug-in. Some requested the ability to have multi-version merges, like in MS Word, where tracking of changes by several users on the same document is supported with different coloured highlighting.

Most existing diagramming tools with versioning support provide model-based differencing using diff or XML diff-based tools [18], [19]. Those that provide diagram differencing utilize textual comparison of diagram content. For example Rational Rose and Magic Draw convert diagrams into a hierarchical textual representation which is diffed and then changes between diagram versions presented as highlighting schemes on the text. The main drawback of this approach is that changes are no longer visible in graphical form and thus more difficult to comprehend (or introduce *hard mental operations* in Cognitive Dimensions terminology) [20]. In contrast our plug-ins provide in-situ presentation of differences within diagrams and interactive accept/reject by users with immediate visualization of the accepted merged change. The generic nature of our differencing algorithm and use of Pounamu Command objects to represent differences between versions means it can be applied to *any* Pounamu-specified diagramming tool; for example, Figure 1 (c) and (d) show application to a Gantt chart tool generated using Pounamu. As the architecture of our plug-ins uses Pounamu's view representation objects and Command objects to update view content on merging, it is compatible with other Pounamu core features and plug-ins. These include semantic constraint checking via event-driven rules, model-view consistency across multiple diagrams when changes are merged into a diagram, and seamless integration with our synchronous editing plug-ins for Pounamu. In addition, because of this architectural approach we have managed to integrate our differentiation plug-in into our Eclipse modeling plug-in for Pounamu, using the Eclipse plug-in's object adaptors without modification of either.

Our approach has some limitations. As indicated above from our user survey and assessment against Gutwin's framework, users cannot control how changes detected by the differentiation plug-in are presented. As Pounamu is a meta-tool allowing very flexible definition of diagrammatic forms and meta-models, this is somewhat frustrating to users. Similarly, we currently only support batch-oriented comparison of one diagram version to one other, without tracking changes in a diagram across multiple versions or supporting several-version diagram merge. Semantic errors can be introduced easily when merging versions e.g. same-named method or class; type-mismatch, invalid association. Currently we allow Pounamu's constraint mechanism, which is driven by event handlers detecting diagram and model object changes, to detect these and present them using the user-specified mechanism in the meta-tool. Ideally semantic conflicts that may be introduced by accepting a change should be indicated in the annotated diagram similar to syntactic changes. Conflicting syntactic changes e.g. one user has deleted shape while another moves it or sets its properties, are detected by our differentiation algorithm. However no attempt is currently made to re-order

changes or indicate to the user that accepting one change may invalidate another. The scalability of our diagram highlighting approach is limited, with a very complex diagram with a large number of changes resulting in very complex, over-lapping highlighting. We need support for users to see a subset of changes and be able to interact precisely with change highlights in views.

Future work includes providing users with the ability to change the highlighting used by the highlighting plug-in. This can be done using Pounamu's meta-tool to specify individual event handler plug-ins (dynamically loaded Java script) for each highlighting scheme but requires re-engineering our highlighting plug-in. Semantic conflict detection and presentation before and during version merging could be supported using Pounamu event handlers that generate events indicating a conflict. Currently these implement a user-defined conflict presentation strategy themselves via arbitrary plug-in Java scripts. All semantic constraints for a Pounamu tool would instead need to be encoded in a uniform way.

8. SUMMARY

We have developed a set of plug-in components for the Pounamu meta-tool that seamlessly support asynchronous diagram versioning, differentiation and merging. Any diagram type defined with Pounamu may make use of these capabilities to compare versions of the same diagram, the differentiation plug-in generating a set of generic Pounamu Command objects to represent the delta between the versions. A reused view highlighting plug-in visually annotates the diagram to indicate the changes between versions and supports interactive, selective accept/reject of changes by the user. By use of a set of adaptor classes our plug-ins provide similar support within Eclipse graphical editors derived from Pounamu meta-tool specifications.

9. REFERENCES

- [1] Conradi, R. and Westfechtel, B. Version Models for Software Configuration Management. ACM Computing Surveys, vol. 30, no. 2, p. 232-282, 1998.
- [2] Eclipse Version Tree plug-in for CVS, <http://versiontree.sourceforge.net/>
- [3] Ellis, C.A., Gibbs, S.J., and Rein, G.L., Groupware: Some Issues and Experiences, CACM, vol. 34, no. 1, 1991.
- [4] Green, T. R. G., Burnett, M. M., A Ko, J., Rothermel, K. J., Cook, C. R., and Schonfeld, J., Using the cognitive walkthrough to improve the design of a visual programming experiment, 2000 IEEE Conf. on Visual Languages, pp. 172-179.
- [5] Grundy, J.C., Hosking, J.G. Mugridge, W.B. and Amor, R. Support for collaborative, integrated software development, 7th IEEE Conf. on Software Engineering Environments, Nordwijkerhout, The Netherlands, 4-5 April 1995.
- [6] Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Inconsistency management for multiple-view software development environments, IEEE Transactions on Software Engineering, vol. 24, no. 11, November 1998, 960-681.
- [7] GNU, CVS - Concurrent Versions System, www.gnu.org/software/cvs
- [8] Gutwin, C. and Greenberg, S., A Descriptive Framework of Workspace Awareness for Real-Time Groupware, Computer Supported Cooperative Work, vol. 11 no. 3, 2002, 411-446.
- [9] Mehra, A. Grundy, J.C. and Hosking, J.G. Adding Group Awareness to Design Tools using a Plug-in, Web Service-based Approach, 6th International Workshop on Collaborative Editing Systems, CSCW, Nov 2004, Chicago.
- [10] Heckel, P. A technique for Isolating Differences Between Files, CACM, vol. 21, no. 4, April 1978, pp. 264-268.
- [11] Hunt, J.W., and McIlroy, M.D., An Algorithm for Differential File Comparison., Computing Science Technical Report No. 41, Bell Laboratories, 1975.
- [12] IBM. XML Diff and Merge Tool. <http://www.alphaworks.ibm.com/tech/xmldiffmerge>.
- [13] IBM, IBM Rational Software, <http://www-306.ibm.com/software/rational/>.
- [14] Magnusson, B., Asklund, U., and Minör, S., "Fine-grained Revision Control for Collaborative Software Development," 1993 ACM SIGSOFT Conf. on the Foundations of Software Engineering, Los Angeles CA, Dec. 1993, pp. 7-10.
- [15] Miller, W. and Myers, E.W., A File Comparison Program. Software - Practice and Experience, vol. 15, no.11, November 1985, 1025-1040.
- [16] No Magic Inc., 9.0 ed: MagicDraw UML, 2005.
- [17] Ohst, D., Welle, M. and Kelter, U. Difference tools for analysis and design documents, 2003 IEEE Conf. on Software Maintenance.
- [18] Sparc Systems, The Compare Utility (Diff), <http://www.sparxsystems.com/resources/diff/>.
- [19] Stylus Studio, XML Diff tool, http://www.stylusstudio.com/xml_diff.html.
- [20] Tam, T., Greenberg, S. and Maurer, F., Change Management, Western Computer Graphics Symposium, Panorama Mountain Village, BC, Canada, 2000.
- [21] Tichy, W. F. 1985. RCS-a system for version control. Software-Practice & Experience, vol. 15, no. 7, 637-654.
- [22] van der Westhuizen, C. and van der Hoek, A. Understanding and Propagating Architectural Changes, 3rd IEEE/IFIP Conference on Software Architecture, pp. 95-109.
- [23] Zhu, N., Grundy, J.C. and Hosking, J.G.. Pounamu: a meta-tool for multi-view visual language environment construction, 2004 IEEE Conf. on Visual Languages and Human-Centric Computing, Rome, 25-29 Sept. 2004, 2004.
- [24] Zou, L. and Godfrey, M. Detecting Merging and Splitting using Origin Analysis, 10th International Working Conference on Reverse Engineering, Victoria, B.C., Canada, 11-13 Nov, 2003.
- [25] Zündorf, A., Wadsack, J.P., and Rockel, I. Merging Graph-Like Object Structures. 10th International Workshop on Software Configuration Management. 2001.

6.4 A 3D Business Metaphor for Program Visualization

Panas, T., Berrigan, R. and Grundy, J.C. A 3D Business Metaphor for Program Visualization, In *Proceedings of the 2003 Conference on Information Visualisation*, London, 16-18 July 2003, IEEE, pp. 314-319.

DOI: [10.1109/IV.2003.1217996](https://doi.org/10.1109/IV.2003.1217996)

Abstract: Software development is difficult because software is complex, the software production process is complex and understanding of software systems is a challenge. We propose a 3D visual approach to depict software production cost related program information to support software maintenance. The information helps us to reduce software maintenance costs, to plan the use of personnel wisely, to appoint experts efficiently and to detect system problems early.

My contribution: Co-designed approach, supervised the visiting PhD student, wrote substantial parts of paper, co-lead investigator for funding for this project from FRST

A 3D Metaphor for Software Production Visualization

Thomas Panas
Software Technology Group
Växjö University, Växjö, Sweden
Thomas.Panas@msi.vxu.se

Rebecca Berrigan
Peace Software
Auckland, New Zealand
rebecca.berrigan@peace.com

John Grundy
Department of Computer Science
Auckland University, Auckland, New Zealand
john-g@cs.auckland.ac.nz

ABSTRACT

Software development is difficult because software is complex, the software production process is complex and understanding of software systems is a challenge. In this paper we propose a 3D visual approach to depict software production cost related program information to support software maintenance. The information helps us to reduce software maintenance costs, to plan the use of personnel wisely, to appoint experts efficiently and to detect system problems early.

KEY WORDS

Program Visualization, Information Visualization, Program Understanding, Reverse Engineering, Software Maintenance.

1 Introduction

Software engineers who are to maintain, extend or reuse a software system must understand it in the first place. Obtaining this understanding of an industrial size system is often a time consuming process since most legacy systems are usually sparsely (or inadequately) documented. Estimates of the proportion of resources and time devoted to maintenance range from 50% to 75% [1, 2]. The greatest part of the software maintenance process, in turn, is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 47% and 62% of time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities [3].

Due to the above reasons, reverse engineering of legacy systems has become a vital matter in today's software industry. However, increasing program understandability is not a simple task due to the complexity of software systems. It is assumed that software complexity is related to the commonly accepted empirically derived features of soft-

ware like code size, coupling, depth of inheritance, McCabe complexity related to logical paths and data flow, amongst others. It is also likely that the *cost* of related aspects in a system such as defect levels or the effort required to design, develop, test and maintain a software product gather more attention in a business context focused on software production.

Therefore, to simplify software complexity and increase program understandability while correlating with cost profiles across the product, reverse engineered programs must be effectively analyzed and visualized. Since developers think and perceive information differently, a tool must support user specific metaphors and views. Currently, metaphors and views within reverse engineering are restricted to depict program specific data only, i.e. they show merely information retrieved from code or data analysis. However, it is important and necessary to establish cost related analyses and views, which give answers to questions like: Which components are never executed? Which components are often changed? On which components do my developers currently work? Is the architectural structure of my system obsolete and hence needs to be refactored? Therefore, within this paper we present an idea that helps system maintainers and managers to e.g. decide, which components in a system are superfluous, which ones cause high cost due to frequent changes and whether the entire system needs to be revised.

In Section 2 we give a brief introduction to program visualization for software maintenance. In Section 3 we present an example, showing a 3D city metaphor for program visualization. On top of this picture, we present a code production focussed metaphor in Section 4. Related work is summarised in Section 5. A discussion and future work is presented in Section 6. Finally, the paper is concluded in Section 7.

2 Program Visualization

Visualization is the presentation of pictures, where each picture presents an amount of easy distinguishable artifacts that are connected through some well defined relations. Visualization itself has a number of specialized foci [4]. However, in this paper we are merely interested in program visualization, dealing with static and dynamic code visualizations.

Within program visualization, the scale and complexity of software are pressing issues, as is the associated information overload problem that this brings. In an attempt to address this problem the following concepts are considered to be important [5]:

- **Abstractions** Program representations resulting from program analysis contain already for middle size programs enormous quantities of information. Consequently, to be able to adequately represent and comprehend the most relevant program information, the information assembled needs to be focused. There are basically three techniques to focus information [6]: Information abstraction, compression and fusion.
- **Metaphors** The mapping from a program model (lower level of abstraction) to an image (higher level of abstraction) is defined through a metaphor, specifying the type of visualization. Most visualization techniques and tools are based on the graph metaphor (including the extensive research on graph layout algorithms). Other initiatives are the representation of programs as 3D city notations [7], solar systems [8], video games [7, 9], nested boxes [10, 11], 3D Space [12], etc.
- **Visualizations** It is not feasible to depict all kinds of program model phenomena in just one picture when the model carries too much information. Therefore each program model is depicted through various views, guaranteeing that the right subsets of objects and their relations are depicted and understood. Most effort to solve software complexity was put into different visualization forms, mainly the graph metaphor, including UML diagrams, to depict various program class and architecture views [4, 13, 14, 15].

The success and quality of any visualization depends on many vital features [4]: Animation, Metaphors, Interconnection, Interaction, Dynamic Scale. However, most vital for successful program visualization is the retrieval of necessary data for visualization and the availability of a suitable metaphor.

3 A 3D City Example for Program Development Visualization

Metaphors, when depicting real worlds and establishing social interaction [16], especially in virtual reality [17, 18], become very important. Essential is therefore the choice of metaphor to improve the usability of a system. One fundamental problem with many graphic designs is that they have no intuitive interpretation, and the user must be trained in order to understand them. Metaphors found in nature or in the real world avoid this by providing a graphic design that the user already understands. When illustrating a reverse engineered architecture, it is important for the understanding of a program that the final picture is adjusted for the individual [19, 20]. Therefore, we are currently developing a visualization architecture that allows all kind of metaphors to increase individual program understandability.

Within our unified recovery architecture [6, 21], the selection of metaphor can be undertaken depending on the user's requirements and focus in visualizing program information. Currently, we are implementing a 3D City metaphor to our architecture to support program understanding within three dimensions - this could be changed for an alternate world in a user dependent fashion. Figure 1 shows a screenshot of the running example, where buildings denote components (mainly Java classes) and the city itself represents a package. Different metaphors between the source code and the visualization are possible, i.e. components must not always be mapped to buildings and packages to cities. Other compilations are thinkable, where e.g. buildings are mapped to methods. To support the user with an intuitive interpretation of a software city and to increase the overall realism of the metaphor, we added trees, streets and street lamps to the figure.

Further, the figure illustrates both, static as well as dynamic information about a program. From a static point of view, the size of the buildings give the system maintainers an idea about the amount of lines of code of the different components. The density of buildings in an certain area shows the amount of coupling between components, where the information for this can easily be retrieved from metric analysis. The quality of the systems implementation within the various components is visualized through the buildings structures, i.e. old and collapsed buildings indicate source code that needs to be refactored.

From the dynamic standpoint, cars moving through the city indicate a program run. Cars originating from different components leave traces in different colors, so that their origin and destination can easily be determined. Dense traffic indicates heavy communication between various components. Performance and priority are depicted through the speed and type of vehicles. Occasionally exceptions occur, where cars collide with other cars or buildings, leading to



Figure 1. 3D City

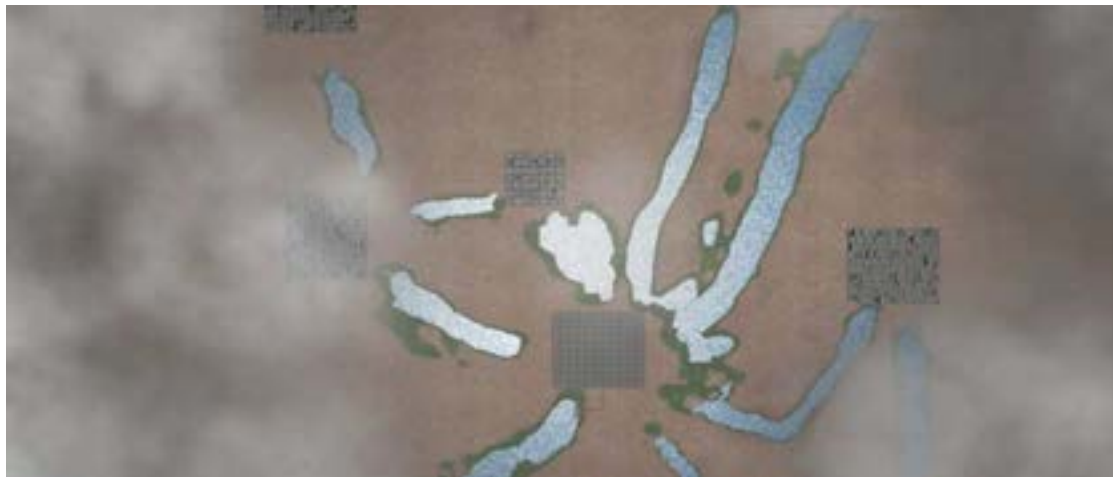


Figure 2. 3D City from Top

explosions.

A satellite view of our analyzed system can be seen in Figure 2, where cities (packages) within the architecture are connected via streets (two-directional calls) or water (unidirectional calls). Information between cities (packages) is passed via boats and vehicles. Again, dynamic as well as static information is illustrated. Clouds in that figure cover cities that are not of current interest to the user and hence hidden.

The general idea is to fly interactively through a reverse engineered software system and depict it within a 3D city. The user must have full freedom in zooming and navigating through the system, and even be able to perceive the system not only on the usual computer monitor, but also within a virtual environment. However, the figures represent, so far, only static and dynamic information about a reverse en-

gineered software system, which is received from the programs source code. For maintenance, however, we wish for more information to answer cost related questions, like where does the main maintenance costs occur in the system. In order to answer this questions in a picture, we present next a software production cost related 3D city metaphor.

4 A Metaphor Designed to Highlight Production Information

Program visualization can provide a large reduction in effort associated with program understanding. However, in order to increase the reactivity of a development environment with respect to business realities, highly specialized, visual information should be delivered in a timely fashion to those people that can effect greatest impact.

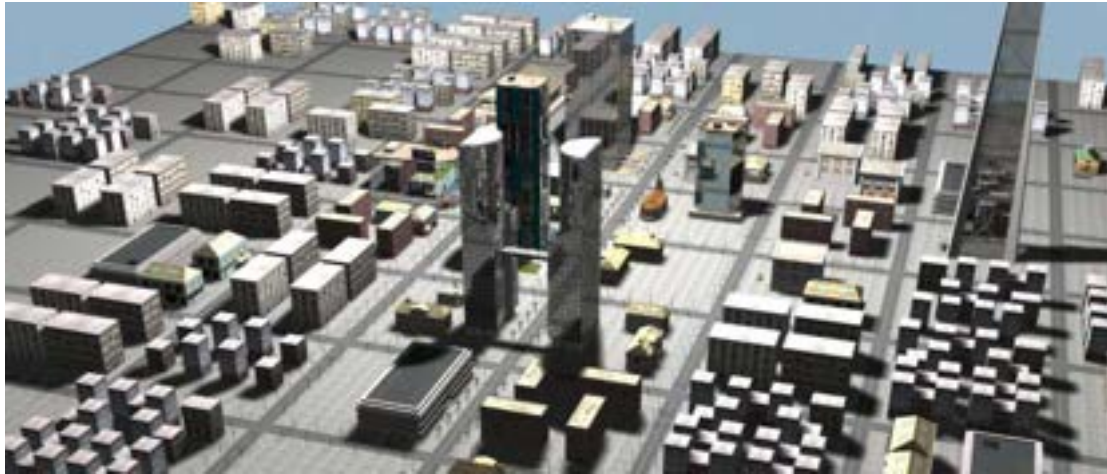


Figure 3. 3D City from Top without Business Info



Figure 4. 3D City from Top with Business Info

Developers are mainly interested in information about functional and non-functional issues of a program, e.g. which components do I have to modify in order to change the security aspect of my system. Project managers, need to know to which parts of the program his team is allocated to, answering questions like, can we meet the next deadline? Additionally, managers and vendors are interested in hot spots, which means components that have been modified frequently, indicating the high cost areas of the systems maintenance. Designers and maintainers are more interested in the overall structure of the system, indicating places, which are of heavy or little use. This helps to decide which components to remove or how to restructure the system in order to achieve a higher performance or reliability.

Although, Figure 2 and Figure 3 help to visualize various static as well as dynamic aspects of a reverse engi-

neered system and hence increase the understandability of the system, they do not provide support for the various additional demands that developers, designers, vendors and project managers have. Therefore, we apply a cost focused metaphor over the 3D city metaphor, to visualize additional business related information of a system. Figure 4 shows our approach.

In the figure we illustrate various additional aspects relevant for different types of stakeholders.

- *Work Distribution.* The components currently being modified by the staff are indicated in yellow with the respective names. This gives an idea about the progress of the maintenance or development team and can help to estimate the total time needed.
- *Recycling.* In brown, those parts of the system are

illustrated, which are no longer needed. These parts have not been used for quite a while and increase only the total complexity of the system. Those parts should be further investigated for removal.

- *Hot Execution Spots.* Components with frequent execution are indicated by a surrounding fire. In order to increase the reliability and performance of a system it is beneficial to investigate these components carefully.
- *Aspects.* Aspects, from Aspect-Oriented Programming [22, 23], can be depicted in various colors. In the figure here, distribution is shown in blue and synchronization in green. These spots help out to point out functional or non-functional cross-cuttings that can be investigated and further changed by experts.
- *High Costs.* Buildings with flashes indicate frequent component modifications. The result is an increased maintenance cost and hence also an increased cost for the entire software project.

The list above is not complete. Many more production related issues exist that could be depicted. Vendors and managers might see the quality of a software system at once, by having a short look at the 3D city. Lots of fire, flashes and mud indicate high cost areas of code and unacceptably high risk regarding the ongoing health of the system.

Developers might benefit from business process, use-case, control flow, data-flow, and event-based visualizations. For example, a dynamic picture of events occurring on various positions in the city and being handled on other positions (indicated by e.g. colors) can aid the understanding of event-based software development. Another example might be an animation of cars driving between the different cities, marking them in color, to indicate which components are used when and for which use case.

Further, static and dynamic source code analysis can be illustrated, showing a changing city, depending on the type of analysis applied. For example, when applying the concept analysis, which computes maximum sets of objects using the same set of attributes, different components can be formed, depending on the user interaction on the concept lattice. For reverse engineering, component detection can be optimized by interactively illustrating the changing picture of the city while the concept lattice of the analysis is being changed.

Finally, the navigation within the city is important in order to present the 3D city to the user in the most natural way. Software maintenance should not necessarily require reading millions of lines of code. Navigating through the city from top or even from inside a car (program counter), might make maintenance a game. Complex, huge software systems could be comprehended much faster.

5 Related Work

At present there exists some work on visualizing source code in three dimensions, however little effort was made to depict production relevant information in 3D. Knight and Munro [7] describe what they call the 'software world' - a system being visualized in the world. Cities represent source files that may contain one or more classes which themselves are represented as districts and finally methods are shown as buildings. Here colors are used on buildings to encode the type of methods: public or private. However, the focus of this paper is the application of virtual reality technology to the problem of visualizing data artifacts and not production related visualization.

In [24] the authors present a case study, which shows the possibility of deploying 3D visualizations in a business context. In the case study a reusable collection of behaviors and visualization primitives is presented (3D gadgets), with the intention to deploy visualization components in a software architecture. However, the paper focuses on business process visualization to validate requirements and to acquire feedback on design-tradeoffs and not on program understanding or software maintenance.

6 Discussion and Future Work

As mentioned above, the 3D city is an example metaphor for our unified visualization architecture, that is currently being developed. The architecture supports any kind of reverse engineering, following three basic steps: analysis of a program, focusing (which means filtering, folding, fusing, etc. of program information), and the visualization of retrieved source code information. For the visualization, a mapping between the data representations and the final picture is needed. Hence, we need metaphors.

Currently, we are about to finish our unified recovery architecture that allows us to plug in various visualizations, including metaphors and layout algorithms, providing the best fit system view for an individual user. The step to follow is to implement the 3D city metaphor in e.g. OpenGL4Java, in order to perform tests and evaluations. The present study about the 3D city is based on our 3DStudio Max implementation.

The unified recovery architecture permits the visual correlation of code structure information and development house operational information in a single viewscreen. The choice of metaphor can also be altered depending on the relative importance of each aspect of the code (or cost related to the code) to the viewer at hand. Once these tools are robust and readily configurable the test will be to undertake a medium scale industrial trial where scalability, performance and time constraints will test the ability of the visualization tool to provide valuable information to the organization.

7 Conclusion

In this paper, we have illustrated our 3D city metaphor that we are currently developing for our unified recovery architecture. Further, we have discovered the need to depict more than just static or dynamic program information for increasing a programs understandability and maintainability. We need to depict production cost related information of a system that shows the right stakeholder the right information using minimal time and effort. In general we have tried to present a vision where software maintenance must not always be labor intensive, trying to understand and alter undocumented legacy code, but also fun, when navigating through a software city in 3D.

References

- [1] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- [2] B. Lientz, E. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6), June 1978.
- [3] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings GUIDE 48, Philadelphia*, April 1983.
- [4] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization*. MIT Press, 1998.
- [5] C. Knight. *Visual Software in Reality*. PhD thesis, University of Durham, 2000.
- [6] T. Panas, W. Löwe, and U. Aßmann. Towards the unified recovery architecture for reverse engineering. In *Int. Conf. on Software Engineering Research and Practice, Las Vegas*, June 2003.
- [7] C. Knight and M. C. Munro. Virtual but visible software. In *IV00*, pages 198–205, 2000.
- [8] P. Damien. Building program metaphors. In *PPIG'96 Post-Graduate Students Workshop at Matlock, UK*, September 1996.
- [9] K. Kahn. Drawing on napkins, video-game animation, and other ways to program computers. *Communications of the ACM*, 39(8):49–59, 1996.
- [10] J. Rekimoto and M. Green. The information cube: Using transparency in 3d information visualization, 1993.
- [11] S. P. Reiss. An engine for the 3d visualization of program information. *Visual Languages and Computing (special issue on Graph Visualization)*, 6(3), 1995.
- [12] J. Rilling and S.P. Mudur. On the use of metaballs to visually map source code structures and analysis results onto 3d space. In *Ninth Working Conference on Reverse Engineering (WCRE'02)*. IEEE, October 2002.
- [13] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4), 1996.
- [14] C. Lewerentz and F. Simon. Metrics-based 3d visualization of large object-oriented programs. In *1st Int. Workshop on Visualizing Software for Understanding and Analysis*, June 2002.
- [15] P. Eades. *Software Visualization*. World Scientific Pub Co, 1996.
- [16] C. Russo dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J.P. Paris. Metaphor-aware 3d navigation. In *IEEE Symposium on Information Visualization*, pages 155–65. Los Alamitos, CA, USA, IEEE Comput. Soc., 2000.
- [17] G. Fitzpatrick, S. Kaplan, and T. Mansfield. Physical spaces, virtual places and social worlds: A study of work in the virtual. In *CSCW'96. ACM Press*, 1996.
- [18] K. Vaananen and J. Schmidt. User interfaces for hypermedia: how to find good metaphors? In *CHI '94 conference companion on Human factors in computing systems*, pages 263–264. ACM Press, 1994.
- [19] J.F. Hopkins and P.A. Fishwick. The rube framework for personalized 3d software visualization. In *Software Visualization. International Seminar. Revised Papers (Lecture Notes in Computer Science Vol.2269)*. Springer-Verlag, pages 368–380. Berlin, Germany, 2002.
- [20] S. North. Procession: using intelligent 3d information visualization to support client understanding during construction projects. In *Proceedings of Spie - the International Society for Optical Engineering*, volume 3960, pages 356–64. USA, 2000.
- [21] VizzAnalyzer. <http://www.msi.vxu.se/~tps/VizzAnalyzer>, 2003.
- [22] G. Kiczales, K. Lieberherr, H. Ossher, M. Aksit, and T. Elrad. Discussing Aspects of AOP. *Communications of the ACM*, 44(10), October 2001.
- [23] T. Panas, J. Andersson, and U. Aßmann. The editing aspect of aspects. In I. Hussain, editor, *Software Engineering and Applications (SEA 2002)*, Cambridge, November 2002. ACTA Press.
- [24] B. Schönhage, A. van Ballegooij, and A. Eliens. 3d gadgets for business process visualization - a case study.

6.5 Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool

Grundy, J.C. and Hosking, J.G. Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool, In *Proceedings of the 2007 IEEE/ACM International Conference on Software Engineering (ICSE'07)*, Minneapolis, USA, May 2007, IEEE CS Press, pp. 282-291.

DOI: [10.1109/ICSE.2007.81](https://doi.org/10.1109/ICSE.2007.81)

Abstract: Software engineers often use hand-drawn diagrams as preliminary design artefacts and as annotations during reviews. We describe the addition of sketching support to a domain-specific visual language meta-tool enabling a wide range of diagram-based design tools to leverage this human-centric interaction support. Our approach allows visual design tools generated from high-level specifications to incorporate a range of sketching-based functionality including both eager and lazy recognition, moving from sketch to formalized content and back and using sketches for secondary annotation and collaborative design review. We illustrate the use of our sketching extension for an example domain-specific visual design tool and describe the architecture and implementation of the extension as a plug-in for our Eclipse-based meta-tool.

My contribution: Developed initial ideas for this research, co-designed approach, wrote and evaluated the software, wrote majority of paper, co-lead investigator for funding for this project from FRST

Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool

John Grundy^{1,2} and John Hosking²

¹Department of Electrical and Computer Engineering and ²Department of Computer Science
University of Auckland, Private Bag 92019, Auckland 1142, New Zealand
{john-g, john}@cs.auckland.ac.nz

Abstract

Software engineers often use hand-drawn diagrams as preliminary design artefacts and as annotations during reviews. We describe the addition of sketching support to a domain-specific visual language meta-tool enabling a wide range of diagram-based design tools to leverage this human-centric interaction support. Our approach allows visual design tools generated from high-level specifications to incorporate a range of sketching-based functionality including both eager and lazy recognition, moving from sketch to formalized content and back, and using sketches for secondary annotation and collaborative design review. We illustrate the use of our sketching extension for an example domain-specific visual design tool and describe the architecture and implementation of the extension as a plug-in for our Eclipse-based meta-tool.

1. Introduction

Hand-drawn sketches are often used in software engineering across many phases in the software development process. These include high-level requirements capture, system design, user interface design and code review [2,26,29]. A variety of increasingly popular hardware devices support sketch-based input to computer applications, including the Tablet PC, mobile PDAs, large-screen E-whiteboards, and plug-in tablets for conventional PCs and laptops. Much recent research in HCI and user interfaces has demonstrated the potential of such sketching-based user interfaces to enable more human-centric interaction with computers and to enhance the efficiency and effectiveness of user interfaces, particularly for early-phase design and during collaborative work [5,8,11,15,16,29].

However, most existing software engineering tools lack support for sketching-based input, with the exception of informal annotation in a few tools, e.g. [7]. A small number of design-oriented applications have attempted to provide sketching-based UML and user interface design support [5,8,9,20,28], and a few applications have leveraged sketching-based input for code review and to facilitate communication for collaborative work support

[8,16,30]. However all of these systems use either special-purpose tool implementations with limited functionality and integration support or ad-hoc techniques to add sketching-support into existing tools and provide very limited user control over the recognition and formalization of sketched content.

We have developed a meta-tool for building a wide range of domain-specific visual language tools for software engineering design tool development and other diagrammatic modelling applications [14,33]. We have also developed stand-alone, ad-hoc sketching support for early-phase UML design [5]. The success of the latter suggested the usefulness of adding sketching features into our design tool meta-toolset. This would allow any diagram-centric design tool generated by the meta-tool to provide flexible sketching-based input. Given the generality of the meta-toolset, we wanted to provide users with flexible control over the approaches used for sketched content input and processing. To achieve this, we have enhanced our meta-tool, which is realised as a set of Eclipse plug-ins, with an extra plug-in to support flexible sketch-based input in any generated tool implementation. The support provided includes both eager and lazy shape recognition; progressive formalization of sketches into computer-drawn content; preservation of sketched content; and the ability for users to easily move between sketched diagrams and formalized diagram content, or even to mix the two. Our generated design tools with sketch-based input run as Eclipse plug-ins. While our sketching support currently only works for our Marama-implemented tools the use of Eclipse does allow close integration with other Eclipse-based software engineering tools.

We first introduce a motivation for this research and identify a set of key requirements for generic software tool sketch-based input support. We survey related research in this area and outline the main features of our approach, providing an example illustrating the use of our sketching-based design tools. The architecture and implementation of our meta-tool and the additional sketching plug-ins are described. We finish with discussion of our experiences with these tools and their strengths and weaknesses and key areas for future research.



Figure 1. Examples of software design with MaramaMTE.

2. Motivation

Consider the design of a complex software architecture. Engineers often use multiple views to capture key architectural requirements, key architectural abstractions, and structural vs. behavioural aspects of architectural components [18]. In doing so, they often sketch out preliminary designs for the system architecture, refining the views as design progresses. They may review their designs collaboratively with other engineers and developers before moving into detailed design and implementation of the architecture using a variety of design and coding tools.

We have developed several tools to support such architectural capture. One of them, MaramaMTE [10,13], is shown during use in Figure 1. Figure 1 (a) is a structural diagram showing client, server, database and other key structural abstractions. Figure 1 (b) shows a model of client behaviour as pages, actions and inter-relationships. Such an architecture may be refined to or reverse engineered from a set of more detailed UML design diagrams, e.g. Figure 1 (c), and user interface designs, e.g. Figure 1 (d).

During development of MaramaMTE-like design diagrams, and related UML and user interface designs, software engineers may want to use sketches to assist their design conceptualisation. Using hand-drawn designs in this way has the demonstrated advantages over conventional software diagramming tools of flexibility, encouragement of exploratory design, and support for collaborative annotation and design review [2,26,29]. Conventional

diagramming tools have been shown by many studies to suffer from premature commitment to particular design artefacts and choices; over-constraint of user actions; high viscosity (making designs hard to change); stifling of creativity; and limited collaborative design and review support [29]. As many studies have shown, sketching-based design tools for a wide range of diagram-centric tasks offer ways of combining the advantages of paper-based and whiteboard-based design with those of computer-based diagramming tools [2,5,7,11,15,20,29]. To date, though, most such research has focused on supporting sketching-based input in a narrow range of design tasks i.e. only providing for very limited diagram types, and in ad-hoc ways i.e. strongly tied to one tool with very limited reusability of sketching and recognition support.

To more widely realise the advantages of sketch input we were motivated to add effective sketching support to all diagram-based design tools realised using our Marama meta-toolset. Marama supports the specification and generation of Eclipse-based design tools, including the MaramaMTE, UML and XForm design tools in Figure 1. Marama tool specifications, designed using a set of primarily visual meta-tools, are loaded by a set of Eclipse plug-ins which generate each diagramming tool. In order to add sketch-based design support for Marama tools we identified the following key requirements:

- *Sketching for any Marama tool.* Sketch-based input, recognition and annotation for any Marama-generated diagramming tool must be supported. Users should be able to “draw”, using a mouse, tablet PC stylus, external tablet or Mimio-style E-whiteboard pen,

content that is captured by the Marama design tool. The sketched diagram elements should then be recognised and converted into Marama diagram elements. There should be minimal (or ideally no) modification or extension of the Marama diagramming tool specifications to support sketching-based input.

- *Flexible recognition and conversion.* Diagramming tool users need flexible control over when content is recognised and converted i.e. eager vs lazy sketched shape recognition. Some diagramming tasks (and user preferences) suit conversion of a sketched shape into a computer-drawn Marama shape immediately it is drawn. Many others better-suit sketching a whole diagram and recognising and converting it as a whole. Still others suit a mixture of approaches, particularly annotation of a conventionally-edited diagram for design review tasks.
- *Recognition accuracy.* Accurate shape and text recognition has been shown to be essential for sketch interfaces [22]. However, users should be provided with the ability to very easily over-ride the recogniser when it makes an error. The recogniser may need to support training to individual user sketching styles [17, 27, 31].
- *Seamless movement.* Users need to be able to move easily between sketching-based diagram input and annotation and conventional mouse-driven editing of diagram content within the IDE.
- *Collaboration support.* Collaborative design and review support should include distributed, multi-user sketch-based diagram input and annotation.

3. Related Work

Many CASE tools support UML modelling, almost all using conventional mouse/keyboard input and formalised icons [32]. Industry adoption of these tools has been mixed [2,19] with empirical studies showing designers find them to be overly restrictive during early design with developers preferring to sketch early designs by hand [2,8,11,19,27]. Diagram editing constraints can also be very distracting to users, especially during creative design work [3,19].

There has been considerable work in the area of pen based sketch input of software designs, with support for formalization of sketches into design artifacts. One of the earliest, SILK [20], allows software designers to sketch an interface using an electronic pad and stylus. SILK recognizes widgets and other interface elements as soon as they are drawn and can transform sketches into standard Motif widgets. Denim [21] is a similar approach to SILK but for web interface design.

Freeform [28] provides sketch definition and testing of Visual Basic forms. Freeform user studies shows that providing interaction capability with retained sketches encourages more complete exploration of design

alternatives. Forms3, a spreadsheet style end user programming environment, has been extended with gestural input [4]. Amulet supports gesture-based document manipulation [25] while Knight [8], SUMLOW [5], and Donaldson et al [9] support UML diagram sketching. Most immediately convert sketched input into computer-drawn formalized content. However, user evaluations for SUMLOW showed that keeping sketched designs is very effective during early phase UML diagramming and when collaboratively reviewing and revising designs with an E-whiteboard. PenMarked provides pen-based code annotation support [30]. Its user studies showed good efficacy of retained pen annotations for code review. These various systems and user studies have affirmed to us that preserving sketch content and having it formalized in flexible ways is both appropriate and useful to support effective software design and review.

Each of these systems is, however, closely tied to the underlying tool it is providing sketch recognition for with little attention to reuse for other sketch-based applications. A variety of low level sketch support tools have been designed with reuse in mind. These include Rubine's [31] single stroke gesture recognition algorithm (used by SILK and Freeform) and Apte's [1] multi-stroke algorithms. Hse has developed the multi-stroke recognition approach into HHReco, a reusable Java toolkit supporting sketching which incorporates a range of trainable and customizable recognisers [17]. While these toolkits are all immensely useful, they still require significant programming to incorporate into other applications. Our interest in this work was in making such generic sketch support available to a wide range of design tools without modification or additional programming on a tool-by-tool basis.

4. Our Approach: MaramaSketch

In order to develop a sketching-based extension for Marama diagramming tools we developed a new plug-in, MaramaSketch. This provides an overlay for Marama diagrams allowing sketching-based input and manipulation of diagram content along with associated shape and text recognition support. Figure 2 illustrates the process of using MaramaSketch.

A tool developer uses the Marama meta-tools to specify a design tool (1). A set of core Eclipse plug-ins provides diagram and model management support for Marama modelling tools. A tool user opens or creates a new modelling project and diagrams using these plug-ins. If installed, an additional MaramaSketch plug-in augments Marama diagramming editing with sketch-based input and recognition (2). When a diagram is created or opened in Marama, a sketching "layer" is created and managed by the MaramaSketch plug-in (3). This intercepts mouse/pen input on the diagram canvas when the MaramaSketch input tool is selected by the user. Drawing with the sketch input

tool creates sketch layer elements (single-stroke and multi-stroke shapes) (4). Depending on user preferences, sketched input may be: immediately recognised and converted to Marama diagram content; recognised but not immediately converted; or converted on-demand by the user e.g. after a whole design has been sketched (5). The user may select conventional Marama diagram edit tools and modify the Marama diagram content e.g. move, resize or delete Marama diagram elements. Such edits are propagated back to the sketch elements associated with the Marama diagram elements (6). Collaborative editing and review are supported using a further plug-in component.

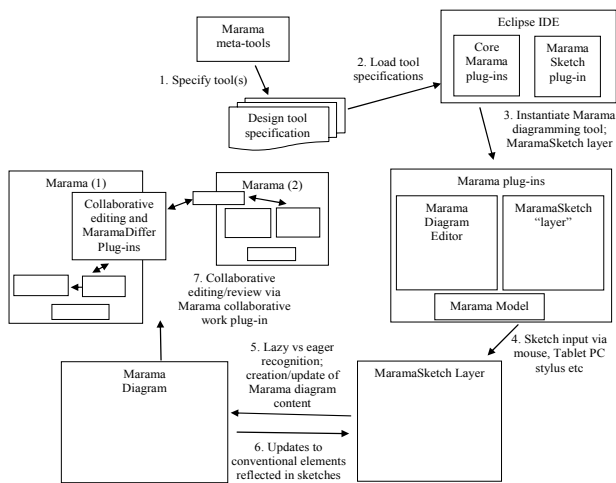


Figure 2. Using MaramaSketch.

5. Example Usage

We illustrate MaramaSketch’s capabilities via its use with the MaramaMTE architecture design tool shown in Figure 1. Initially a tool developer specifies a diagram-based design tool using a set of visual meta-tools [14]. Figure 3 shows part of the meta-tool definition for the MaramaMTE architecture design tool. Shown is a shape specification (a) - this one is for an *ApplicationServer*, and (b) part of the view type specification (set of shapes and connectors and their relationship to an underlying model) for the *ArchitectureView* diagrams. This Pounamu tool specification is loaded by Marama when requested by the Marama tool user. It provides the available diagram elements that can be input by a tool user and thus that may need recognition from sketched input.

A crucial aspect of the success of sketching-based design tools is accuracy of the shape and text recogniser(s) employed [22]. We chose to use a multi-stroke, training-based shape and text recognition algorithm [17] for MaramaSketch. This was primarily to allow individual users to describe their own examples of each available shape type for MaramaSketch to increase the accuracy of its recognition. Earlier work that we did with a non-

trainable recogniser for UML diagramming had insufficient accuracy which became frustrating to users [5]. In addition, as MaramaSketch is intended to support any kind of Marama diagramming tool the available shape types are virtually infinite, leading to eventual difficulty distinguishing between both simple and complex shapes if a non-tool-specific approach is taken. We decided to provide users with the ability to incrementally re-train their MaramaSketch shape and text recognisers while the tool is in use. When a sketched item is incorrectly recognised the user can over-ride the MaramaSketch-recognised shape and ask for the new shape to be added to the recogniser training set. Users can share their training sets so one user might initially specify available shape examples and other may use these, re-training the recogniser over time.

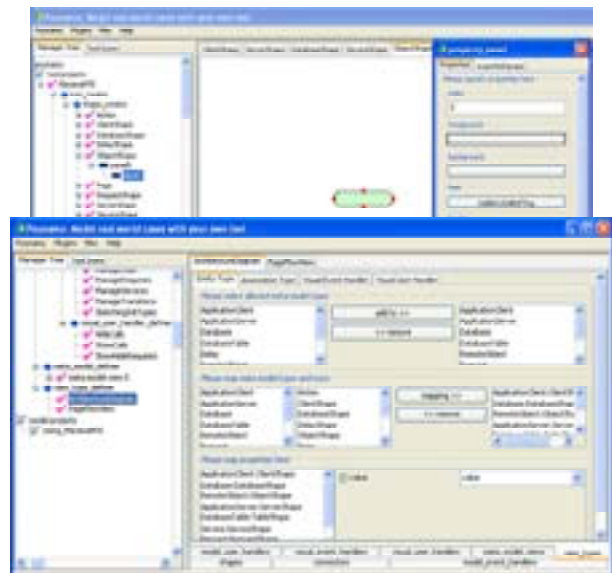


Figure 3. Pounamu tool definition examples.

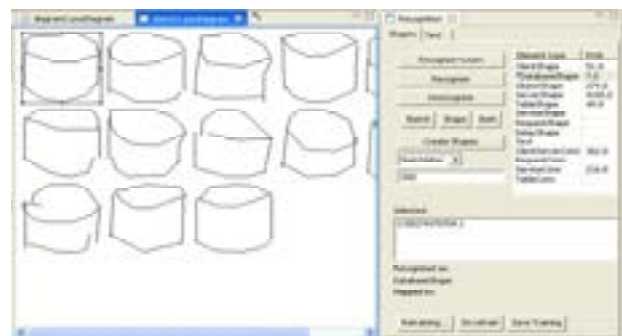


Figure 4. Shape recogniser training example.

Figure 4 shows a user training the MaramaSketch shape recogniser by specifying multiple, multi-stroke examples of a shape. We use various heuristics to identify mouse or stylus strokes as belonging to the same shape, including proximity, time between stroke end/start, and information returned by the recogniser. We use the same

algorithm but different training set and stroke grouping heuristics for text recognition.

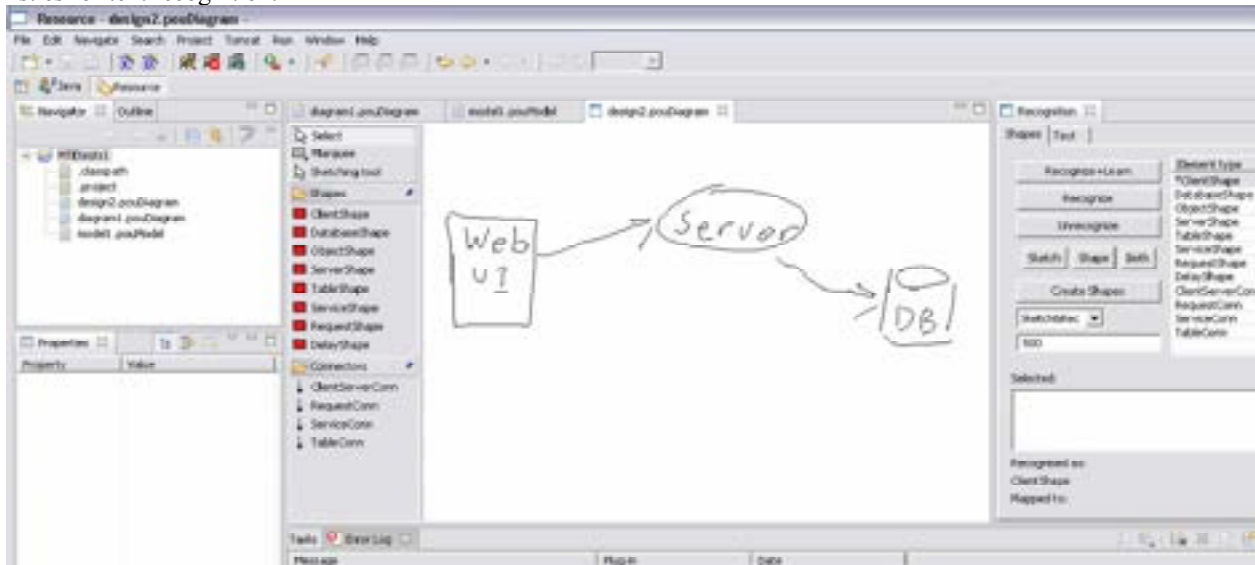


Figure 5. Drawing an architecture design with MaramaSketch.

Figure 5 shows an example of a user drawing content (in this example with a Tablet PC stylus) onto a MaramaMTE *ArchitectureView* diagram. The user simply selects the sketching tool (highlighted in the left hand side editing palette) and draws with the mouse/stylus on the diagram canvas. In this example the user has drawn a *ClientShape* (rectangle, “Web UI”), an *ApplicationServerShape* (oval, “Server”), a *DatabaseShape* (cylinder, “DB”) and two connections between shapes. As each set of strokes is completed MaramaSketch recognises the shape type and remembers this.

The Recogniser view on the right is normally hidden but as illustrated in this example it shows the MaramaSketch shape recogniser probabilities for the most recently drawn or selected and grouped strokes (in this example a new *ClientShape* sketch). The user may override the recognition and learn the new sketched shape as an example of specified shape type via either a pop-up menu or this Recogniser view. Depending on user preferences the drawn strokes can be (1) left unrecognised; (2) recognised as a Marama shape type or text string; (3) recognised and a Marama diagram shape, connector or text property value created; or (4) recognised and immediately replaced by a computer-rendered Marama diagram shape, connector or text property value. Figure 6 shows examples of each of these approaches.

In (1), the user simply draws multi-stroke shapes and Marama doesn’t attempt any recognition or grouping. In (2), the user has asked Marama to recognise and if necessary group strokes. Here, the sketched shape (made up of 4 lines) has been recognised as a *ClientShape* type and the 4 lines grouped into one composite sketched shape. In (3), the sketched shape has been recognised and a

Marama *ClientShape* created and its size and location inferred from the sketched strokes. The user has asked that the sketched strokes and new Marama shapes be shown together (one can be switched off by user preference). In (4), a set of strokes making up a *ClientShape* and the client *name* property have been drawn, recognised by MaramaSketch, and converted into a *ClientShape* with *name* set to “Client1”. The user has asked for the sketched strokes to be hidden immediately after recognition.

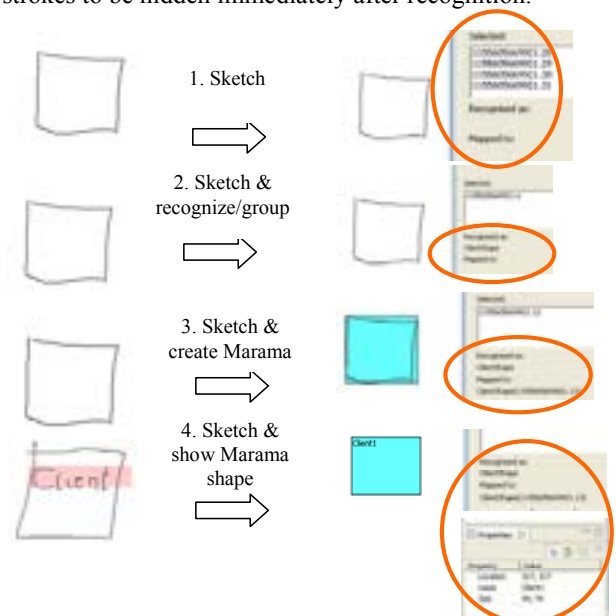


Figure 6. Recognition approaches in MaramaSketch.

A key problem in many sketching-based tools is distinguishing between shapes and text (characters and words). These suit different recognition algorithms and multi-stroke grouping heuristics. We chose to delineate between the two in MaramaSketch by use of a technique we developed for our previous ad-hoc UML sketching tool, SUMLOW [5] and is illustrated in Figure 7. As soon as a sketched shape is recognised as a Marama shape or connector one or more explicit “text area” annotations are automatically added to the sketched shape or connector. These “text areas” (rendered as light pink rectangles) have mouse-over tool-tips indicating the shape or connector property to which the text area corresponds. Any sketched content predominantly inside a text area annotation is assumed to be text and is processed using a different recognition algorithm and stroke grouping heuristics. Again the user can override the recognised text using the Recogniser view or by editing the generated Marama shape property value in the Eclipse Properties view. Text areas can be set to auto-hide after text is recognised, reducing diagram clutter. They can be re-shown for a shape by right-click menu option e.g. to allow over-write modification and then re-recognition of the text.

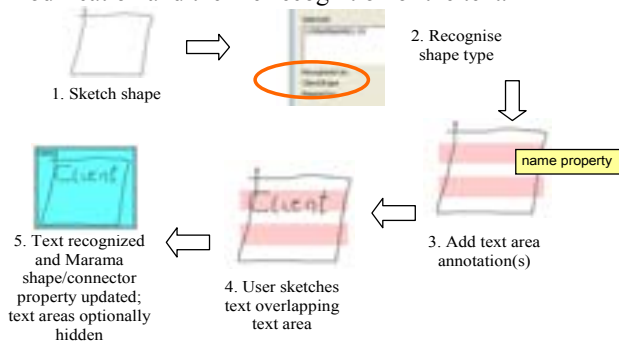


Figure 7. Text recognition in MaramaSketch.

This approach does introduce some premature commitment [12] as shapes must be drawn and recognised before the text annotations can be added otherwise the text

won't be recognised as being associated with shape. In practice, using the approach (2) of Figure 6, this does not prove intrusive to the sketching process as shapes are recognised quickly enough for the annotation areas to be added immediately the shapes are sketched. This still means that shapes must be drawn before text, but this is a fairly natural ordering when sketching iconic shapes so we deemed this limited premature commitment to be appropriate. The user can however force a set of strokes to be recognised as text rather than as a shape or connector by using a right-click menu option.

A similar ordering constraint is currently used when recognising connectors i.e. lines (possibly with arrows and/or other annotations) between shapes. MaramaSketch firstly recognises the source and target shape types and uses these to inform the recogniser of likely connector type from the Marama meta-model for a diagram. This often greatly reduces the possible connector types possible. For example, MaramaMTE client and server shapes can only be linked by a “ClientServerConn” connector type, hence any connector drawn between them must be of this type.

Users can switch between the different recogniser approaches as sketches are drawn allowing a mixture of sketch and formalised diagram elements to appear in the one diagram. Figure 8 (left) shows an example of this. The initial diagram drawn in Figure 5 has been recognised using Approach (3) with formalized Marama shapes and connectors overlaid by the original sketched shapes. An additional *ServiceShape* (rounded rectangle, “Customer Service”) with an embedded *ObjectShape* (rectangle, “Cust”) and two connectors have been added using Approach (2) which leaves the new shapes and connectors in sketched form only. The diagram may be fully formalized at any time. The user may also add a formalised shape directly via the tool palette and one of the shape training examples is added to the sketch layer to represent this (currently just the first training example).

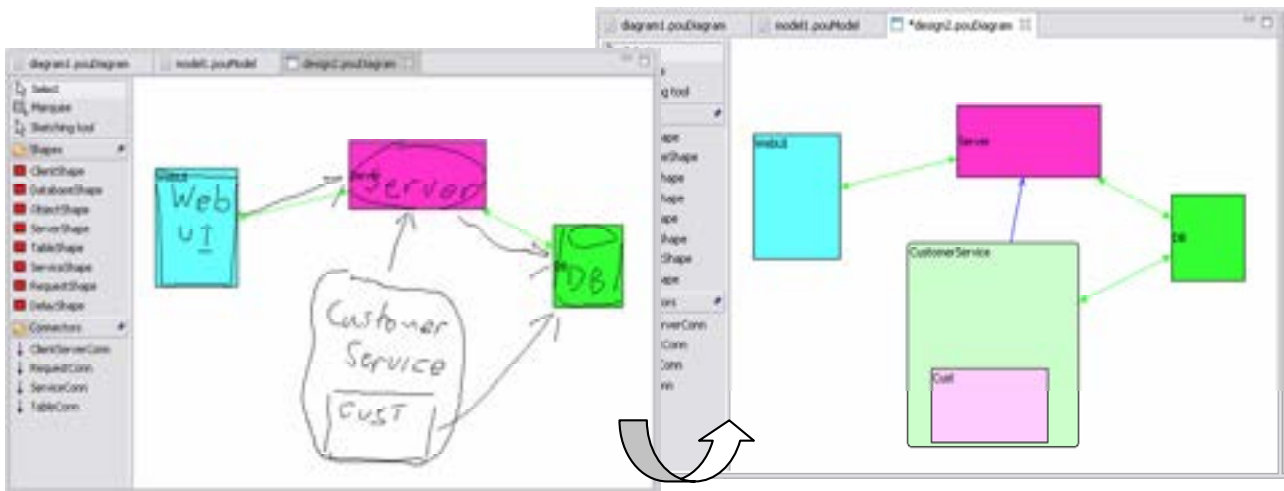


Figure 8. Mixing sketches and Marama diagram elements.

The Cust sketch is recognised as an *ObjectShape* due to its placement inside the *ServiceShape*; only remote objects in MaramaMTE can be placed here. MaramaSketch uses this syntactic information from the Marama diagram meta-model to reduce the possible match options for the shape recogniser and hence improve recognition rates. The fully recognised architecture diagram is shown to the right. The user may freely alternate between the formalised and sketched representations.

Diagrams can have secondary notation added, as is shown in Figure 9. Here a preliminary design is being critiqued with sketched annotations added to capture elements of the design review. These are not recognisable as Architecture Diagram shape types so are ignored by the Recogniser but retained as secondary notation. The user can explicitly stop recognition if desired when doing such “informal annotation” of a diagram.

These annotations also provide an effective collaborative review mechanism where users share these via synchronous or asynchronous editing support. MaramaSketch supports sharing of sketched content via a set of synchronous editing plug-ins we developed for standard Marama diagram synchronous editing [23]. Asynchronous sharing is supported by a shared CVS repository and diagram diffing and merging support, also from Marama plug-ins [24].

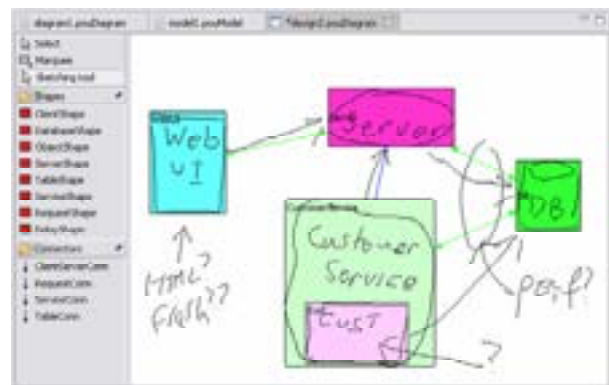


Figure 9. Annotating and reviewing designs.

Figure 10 shows another sketched diagram for a different view type, this time a page flow view. MaramaSketch uses different training sets for each shape/connector type. When a view type is defined as being composed of a particular set of shape and connector types, the matching set of training sets is used by MaramaSketch to recognise shapes in that view type. Thus, although some of the shapes in Figure 10 are similar to those in Figure 9, they are appropriately recognised as page flow elements.

6. Design and Implementation

We developed MaramaSketch on top of our Marama Eclipse-based diagramming toolset [14]. Marama leverages Eclipse’s EMF and GEF frameworks to provide a wide range of diagram editing tools. MaramaSketch extends Marama to provide a sketching layer on top of conventional Marama diagramming tools. The high-level design of MaramaSketch is shown in Figure 11.

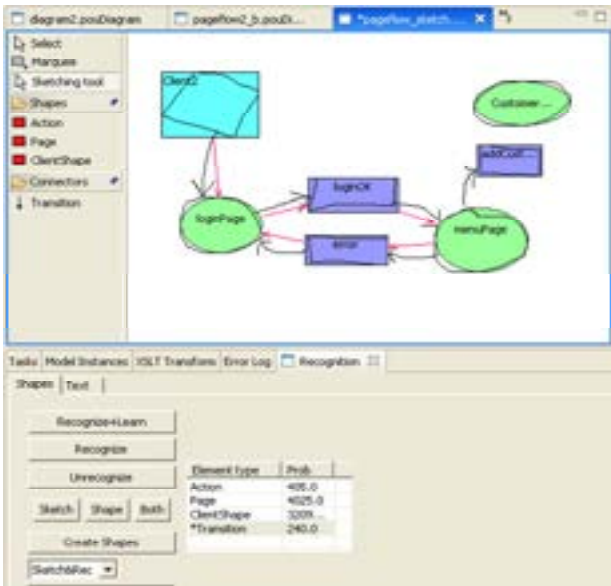


Figure 10. Using MaramaSketch with other view types.

MaramaSketch adds a set of extra view-level components to a Marama diagram: TimedPoints; SketchedShape; and GroupedSketch. TimedPoints capture X,Y co-ordinate and millisecond timing as the user draws on the Marama diagram canvas. The recorded timing information is used by the shape recogniser. TimedPoints are aggregated into a SketchedShape which represents a single stroke shape. Each of these “strokes” may be further aggregated into a GroupSketch, a set of SketchedShapes. A SketchedShape or GroupedSketch may be recognised as and related to a Marama editor shape, connector (line between shapes) or property value (if the GroupedSketch has been recognised as text). The MaramaSketch components form a “layer” above the conventional Marama diagram editor shapes and may be shown or hidden as illustrated previously. They are saved and loaded to the same XMI-format file as the Marama Shape, Connector and Property components for the diagram they have been added to.

Detailed information about Sketched shapes can be viewed and modified via the MaramaSketch Recogniser view, as illustrated previously. This view allows users to explicitly group, ungroup, over-ride the recogniser add new examples to the recogniser and recognise text or graphical shape content. In addition, this augments the Marama editor that has the sketch layer with a set of pop-up menus allowing the user to non-modally over-ride the recogniser, learn new examples, ungroup recogniser-grouped sketched shapes etc.

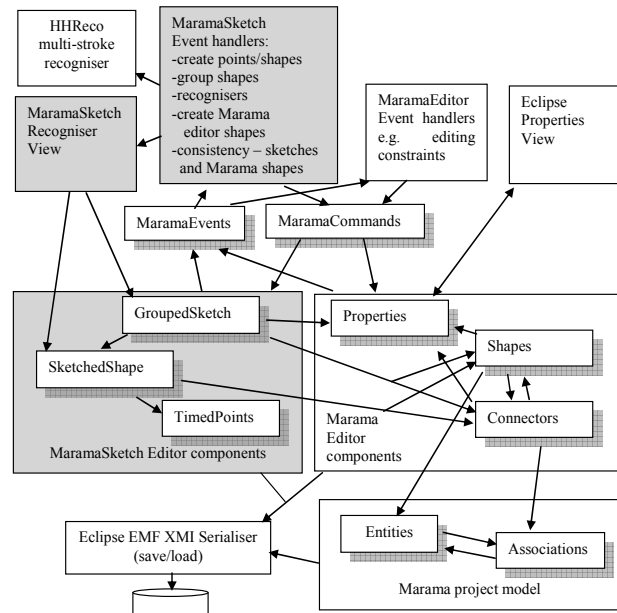


Figure 11. Key architectural abstractions of MaramaSketch.

In Marama, diagram editing constraints and controls are supported via a set of “event handlers” that subscribe to MaramaEvents and describe changes to Marama diagram shapes and connectors. We implemented a set of event handlers for MaramaSketch that listen to changes made to both MaramaSketch layer components and Marama editor components e.g. move mouse, which create TimedPoints, add SketchedShape, resize SketchedShape, move MaramaShape, delete MaramaConnector etc. These event handlers provide the essential MaramaSketch functionality of creating sketched content; grouping sketched shapes; recognising sketched shapes; creating Marama shapes and connectors and relating them to sketched shapes and groups; and modifying sketched shapes when Marama shapes/connectors have been edited and vice-versa. The event handlers make changes to MaramaSketch components and Marama diagram components by creating and running MaramaCommands on them that effect required state changes.

MaramaSketch uses the open source HHReco toolkit to support multi-stroke text and graphical shape recognition [17]. HHReco provides an incrementally re-trainable set of positive and negative examples that can be augmented incrementally during MaramaSketch usage or via custom training sets developed before use. We use two differently configured HHReco recognisers, one for graphical shape recognition and one for textual character recognition. The graphical shape recogniser uses a set of heuristics to group multiple drawing strokes into shapes for recognition. These include time between strokes, stroke overlap, and recogniser probabilities returned when trying different groupings of multiple shapes in sequence.

The HHReco-based text recogniser supports multi-stroke character recognition, in contrast to the common single-stroke approaches such as Rubine’s algorithm [31] and the original Graffiti [22]. Disambiguation as to whether Strokes belong to text vs graphical shapes is currently managed using “text area” annotations, greatly improving recognition rates. Our text recogniser for MaramaSketch could be replaced with e.g. the native text recogniser in the Tablet PC operating system. However this would require running MaramaSketch only on a Tablet PC and would require detailed data structure and API call changes. HHReco allows MaramaSketch to be used on any computing platform with mouse-based input, making the implementation much more portable.

7. Discussion

We have so far used the MaramaSketch plug-in to augment a software architecture design environment (MaramaMTE), a web service composition tool (ViTABaL-WS), a simple UML class diagramming tool, and a music composition tool. No code changes were required for MaramaSketch to work for any of these tools – a single extra event handler is added to the tools’ meta-tool specification to initialise the MaramaSketch capabilities when a diagram is opened. The plug-in has been used on a conventional desktop PC and on a tablet PC and works on either without modification. Due to the prototypical nature of MaramaSketch, in particular the unintuitive user interface provided by the recogniser view, we have not yet conducted an empirical usability study of the plug-in. Instead we have demonstrated the augmented software architecture and web service composition tool to several experienced users and developers of Marama tools and to two novice users of the music composition tool. We obtained preliminary feedback on its potential usefulness in these domains.

Key strengths of the approach we have taken in MaramaSketch include:

- It is generic, working for any Marama tool – even for the design of Marama tools (as our meta-tools are themselves Marama tools). This is in comparison to approaches such as SUMLOW [5] and Knight [8] which are limited to one toolset only.
- It is highly flexible, in that it can be tailored to suit both the tool and end user preferences in terms of its recognition strategy and also in the sketched symbols it will recognise (as embodied in its training sets). Again, this compares favourably to other sketch tools which limit end user choice [5, 8].
- It provides seamless movement both ways between sketching and formalised diagram manipulation.
- It is highly platform portable, limited only by the portability of the underlying Eclipse toolset that it is based on.

Current weaknesses of our approach include:

- The need for training sets. Although they are a key to the tool’s flexibility, they take time to set up when defining a tool. However, this time is amortised over (typically) many applications of that tool definition.
- The user interface is somewhat clumsy when overriding mis-recognised shapes and the prototype recogniser viewer is unintuitive for most users
- The selection of recognition modes e.g. recognise & automatically create shape by users is unintuitive
- The automatic “divider” that determines when to recognise a set of strokes as shape or text is very rudimentary and prone to error
- There is some premature commitment in the approaches taken for text annotation and for connector differentiation, as discussed in the previous section.
- The Marama meta-tool specifications currently have limited information about complex shape relationships e.g. containment and alignment, which if improved would assist shape recognition by reducing options
- It only works for diagramming tools developed using our Marama meta-toolset

The key requirements expressed in Section 2 have all been met. Genericity, flexibility, and seamless movement are described above as key advantages. Recognition accuracy is high and the incremental nature of the training sets means that accuracy can be improved for individual users over time to suit end-user symbol specification preferences.

Premature commitment, which we have discussed in some detail, is one of many dimensions in the Cognitive Dimensions of Notations Framework (CD) [12]. We have used CD to assist us in the design of MaramaSketch. Dimensions we have emphasised, in addition to premature commitment include:

- *Viscosity*: pen and paper/whiteboard sketching has high viscosity; i.e. it takes considerable effort to change a diagram element. In MaramaSketch we have attained much lower viscosity by permitting sketched elements to be resized/moved using the mouse or pen.
- *Progressive evaluation*: we have aimed for high progressive evaluation. End users can have their sketches recognized at any time allowing them to obtain feedback as and when they desire on whether their sketches have been recognised correctly (and can override that recognition if they haven’t)
- *Secondary notation*: again, we have aimed at high secondary notation support. End users can selectively turn off recognition to add any desired form of secondary annotation (or may simply annotate using symbols that are not recognised). This allows arbitrary secondary annotation to be added to any diagram.

- *Closeness of mapping*: this dimension was central to our motivation i.e. that sketching is a more natural mechanism for expressing initial designs than standard computer diagramming approaches.
- *Error-proneness*: The tool currently delineates shapes from text with user assistance (dynamic text areas on shapes) and simple heuristics. While this works if used the way we intended, this approach introduces *premature commitment* and fails if text and shapes are attempted to be recognised in one batch operation.

Other dimensions were less relevant to MaramaSketch's design, as they are more specific to a particular notation/tool implemented by Marama rather than the generic support of MaramaSketch.

There are several areas of improvement that could be made to MaramaSketch. The current implementation uses the HHReco toolkit for text recognition for reasons of portability. Supplementing this with platform specific recognition capability where this is available, such as the Tablet PC text recogniser, would greatly improve recognition performance – particularly for text – at the expense of having to maintain multiple architectures.

An alternative approach to using the text area method for text annotation delineation would be to use a “divider” algorithm, such as is used in the Tablet PC, to automatically infer the distinction between text and graphical objects prior to detailed recognition. This would eliminate the premature commitment issues discussed earlier. The Inkkit toolkit [6] could be used for this purpose. Its divider performance is significantly better than that of the Tablet PC, however it is still platform specific and hence would limit portability.

The current system provides a limited form of “deformalisation” of a standard Marama diagram element i.e. “re-engineering” a sketch from the standard Marama shapes and connectors into realistic-looking sketch elements. As discussed earlier, there is ample evidence to suggest that sketched diagrams encourage designers to explore and critique designs more thoroughly so conversion of formal diagrams into sketches could be useful to encourage that process. MaramaSketch currently provides a limited form of this by selecting the first sketched shape from its training sets to replace formal shape and connector elements in a diagram. These are crudely resized and then combined with similarly generated text annotations. Understanding whether the sketches were then sufficiently realistic to encourage the desired behaviour would then need to be evaluated empirically.

An additional application that MaramaSketch could be extended to is annotation of Eclipse code views. This would use the same sketch overlay mechanism, but to support code annotation and review rather than diagram construction. This would provide a similar mechanism for

Eclipse as Plimmer and Mason [30] have provided for Visual Studio. It may be possible to seamlessly augment any Eclipse GEF (Graphical Editing Framework)-based diagramming tool with a MaramaSketch overlay.

8. Summary

We have described MaramaSketch which generically extends tools generated by our Eclipse-based Marama meta-toolset with sketch input capabilities. The sketching extension is tailorable in its recognition approach, spanning the spectrum from lazy through eager recognition and is incrementally trainable to cope with idiosyncrasies of individual users. Experience with this approach has been promising for providing truly generic sketch input support for software engineering diagramming tools.

References

1. Apte, A. Vo, V. Kimura T. D. Recognizing Multistroke Geometric Shapes: An Experimental Evaluation. In Proc UIST 1993, ACM Press, pp. 121-128.
2. Black, A., Visible planning on paper and on screen: The impact of working medium on decision-making by novice graphic designers. Behaviour and information technology, 1990. 9(4): p. 283-296.
3. Brooks, A. and Scott, L. Constraints in CASE Tools: Results from Curiosity Driven Research, In Proc ASWEC 2001, , 26-28 August 2001, IEEE CS Press, pp. 285-296.
4. Burnett, M. and Gottfried, H Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures, ACM TOCHI 5(1), 1-33, 1998.
5. Chen, Q., Grundy, J.C. and Hosking, J.G. An E-whiteboard Application to Support Early Design-Stage Sketching of UML Diagrams, Proc HCC'03, Auckland, October 2003, 219-226.
6. Chung, R., P. Mirica, and B. Plimmer. *InkKit: A Generic Design Tool for the Tablet PC*. Proc CHINZ 05. 2005. Auckland: ACM: p. 29-30.
7. Churcher N, Cerecke, C, groupCRC: Exploring CSCW Support for Software Engineering, Proc OZCHI'96, 62-68
8. Damm, C.H., K.M. Hansen, and M. Thomsen. *Tool support for cooperative object-oriented design: Gesture based modelling on and electronic whiteboard*. Proc Chi 2000. 2000: ACM: p. 518-525.
9. Donaldson, A. and Williamson, A. Pen-based Input of UML Activity Diagrams for Business Process Modelling, Proc HCI 2005 Workshop on Improving and Assessing Pen-based Input Techniques, Edinburgh, September 2005.
10. Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., Weber, G., Realistic Load Testing of Web Applications. Proc CSMR 2006, IEEE CS Press, 57-70, 2006
11. Goel, V., Sketches of thought. 1995, Cambridge, Massachusetts: The MIT Press.
12. Green, T.R.G. & Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing 1996 (7), pp. 131-174.

7

End-User Applications of DSVLs and MDE

7.1 Domain-specific visual languages for specifying and generating data mapping systems

Grundy, J.C., Hosking, J.G., Amor, R., Mugridge, W.B., Li, M. Domain-specific visual languages for specifying and generating data mapping systems, *Journal of Visual Languages and Computing*, vol. 15, no. 3-4, June-August 2004, Elsevier, pp 243-263,

DOI: [10.1016/j.jvlc.2004.01.003](https://doi.org/10.1016/j.jvlc.2004.01.003)

Abstract: Many application domains, including enterprise systems integration, health informatics and construction IT, require complex data to be transformed from one format to another. We have developed several tools to support specification and generation of such data mappings using domain-specific visual languages. We describe motivation for this work, challenges in developing visual mapping metaphors for different target users and problem domains, and illustrate using examples from several of our developed systems. We compare cognitive dimension-based evaluations of the different approaches and summarise the lessons we have learned.

My contribution: Co-developed initial ideas for much of this research, co-designed approaches, developed and evaluated some of the software, co-supervised Masters student, wrote substantial parts of paper, co-lead investigator for funding for this project from FRST

Domain-Specific Visual Languages for Specifying and Generating Data Mapping Systems

J.C. Grundy, J.G. Hosking, R.W. Amor, W.B. Mugridge, and Y. Li
Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand

Abstract

Many application domains, including enterprise systems integration, health informatics and construction IT, require complex data to be transformed from one format to another. We have developed several tools to support specification and generation of such data mappings using domain-specific visual languages. We describe motivation for this work, challenges in developing visual mapping metaphors for different target users and problem domains, and illustrate using examples from several of our developed systems. We compare cognitive dimensions-based evaluations of the different approaches and summarise the lessons we have learned.

1. Introduction

One of the most common problems in computing is the need to keep multiple views and representations of the same information consistent [13, 16, 32]. Our work in this area has explored consistency management and data mapping approaches in domains as varied as software engineering tools [15], computer integrated construction [2, 3], clinical decision support [16], web services [44], adaptive user interfaces [17], visual notation mapping, and enterprise application integration [6, 18]. Much of this has focused on making consistency management more accessible, via domain-specific languages and tools to allow domain specialists to specify mappings using notations and metaphors appropriate and familiar to end users.

In this paper, we describe several domain-specific approaches to specifying mappings between different data representations, a significant solution component of the consistency management problem. These vary in the sophistication expected of the end user, from skilled programmer through to business analyst. In each case, we have chosen a metaphor for the visual mapping language appropriate to the user, and, in the process, have deliberately limited the expressive power of that language. We begin with an overview of the end user domains and metaphors used, before examining each metaphor in turn for tradeoffs and benefits. We then compare and contrast the solutions we have developed along with those in related work, and summarise the lessons learned from our experiences.

2. Motivation and Overview of Approaches

Complex information structures occur in a wide range of domains. Very often a domain requires the same information to be represented in a variety of ways. This occurs when different organizations have developed tools which use different data formats; when legacy systems and newly-developed systems must be integrated; when different organizations develop competing “standard” representations; and when quite different structures are needed for different tasks, e.g. data analysis versus data visualization [3, 4, 16, 11, 25]. To illustrate the nature of this “data mapping” problem, Fig. 1 shows two example messages representing medical patient treatment information (shown in an XML format, produced by parsing the original surface syntax) [16]. The left message encodes the treatment data using a “deep” structural hierarchy (Patient -> Visits -> Treatments). The right message encodes (mostly) the same data, but uses a flatter format.

To translate the messages we need to apply a variety of field-, record-, and record collection-level translations between these two representations of the patient treatment data. A

number of formulae, some dependent on source message content (e.g. the treatment cost and treatment units), need to be applied. Some structures in the messages repeat, such as the list of treatments required, and these may be organized in quite different ways e.g. a list of Treatments in the left message is grouped into Primary and Other treatments in the right message. To translate the right message into the left, we need to apply data mappings to convert the flat structure into the deeper hierarchical one. To do this several fields and collections must be merged or split e.g. the patient name, dates and address merged.

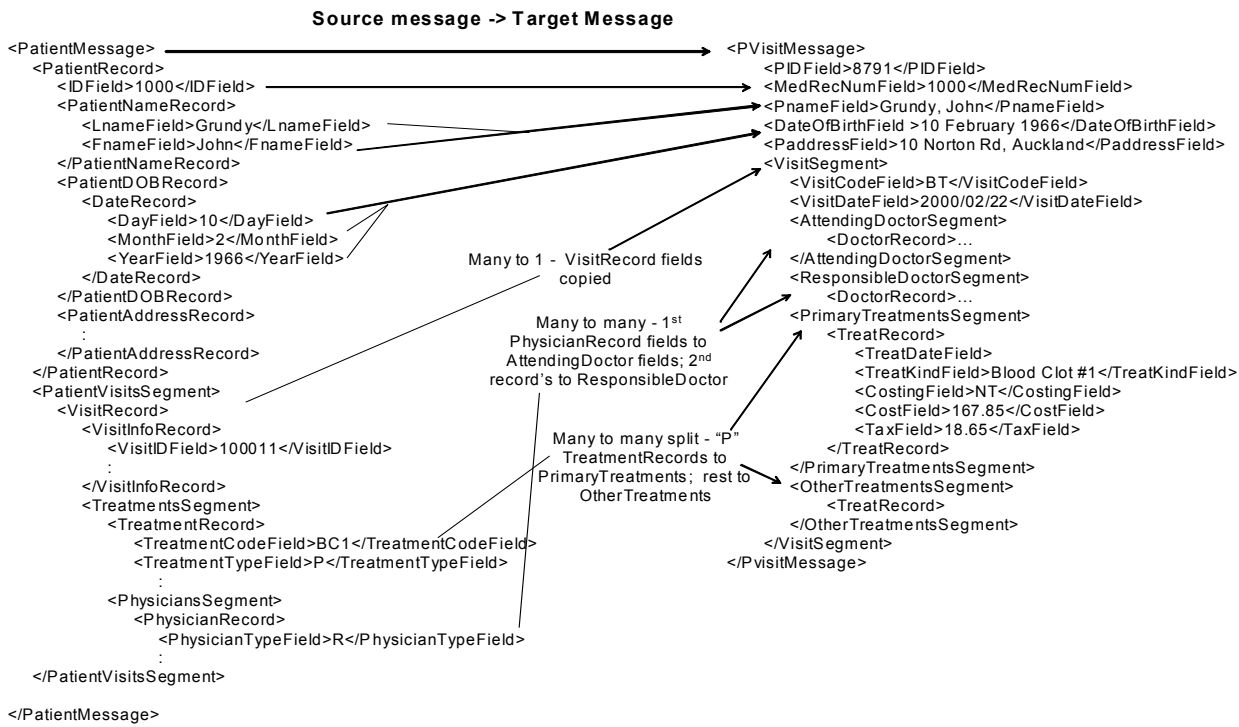


Figure 1. Illustration of mappings between XML formats

This example is typical of the sorts of data mapping problems we have come across in a variety of application domains. However, mappings are usually much larger, often with hundreds of records and fields [2, 6, 16]. This means developers need high-level support to express and manage their message mappings. Typical types of data element mapping [5, 25] include:

- Simple field to field equalities which may involve simple data format or unit conversion, e.g. the `IDField` and `MedRecNumField` in Figure 1 represent the same data.
- Simple formulae relating several fields on one side to one field on the other side, for example the `FnameField` and `LnameField` values are concatenated to form the `PnameField`
- Formulae relating multiple objects or records, possibly spanning parts of a type hierarchy, e.g. the name mapping above maps from a `PatientNameRecord` subrecord of a `PatientRecord` subrecord of a `PatientMessage` record to the `PnameField` of a `PVisitMessage`.
- Formulae mapping collections of objects or records on one or both sides. This could involve 1:n, n:1 or m:n object or record mappings, and involve merging, aggregation, selection, or partitioning of the collections involved. Some examples are shown in Figure 1.

Other mapping issues that need to be addressed include:

- The ability to handle specification of default values, when data is incomplete
- Specification of the directionality of a mapping specification (if not implicit in the language): can the same mapping specification work in either direction?
- Specification of the order in which data elements are mapped (if not implicit in the language, i.e. the flow of control needed to create a complete mapping correctly)
- The need to use meta-data for data formats to assist in defining mapping specifications

- The need to translate data mapping specifications into efficient implementations, and whether these permit mapping of incremental changes from one side to the other (and how efficiently)
- The completeness of the mapping described, enabling a mapping system to determine whether partial information is being transferred.
- The syntax of the data repositories can be intrinsic to the mapping approach or explicitly specified by adjunct processes in the mapping tool e.g. CSV files or EDI messages translated into XML data structures.
- Transaction management for change propagation can be a characteristic of the mapping specification or handled individually within mapping environments.

Specification Language Name	Target User Group	Target Application Domain	Assumed User Programming Ability	Data Structure Visualisation Metaphor	Mapping Visualisation Metaphor
View Mapping Language (VML) [2]	Professional Programmer	Architecture & Engineering design tool integration	High	UML-like class diagrams	Iconic with wired attachments
Rimu Visual Mapper (RVM) [16]	Data Base Administrator	Health Sector Messaging Systems	Moderate	Hierarchical type structure	Mapping segments wired to elements
Form-Based Mapper (FBM) [26]	Business Analyst	Office automation	Low	Business forms	Wires between form elements

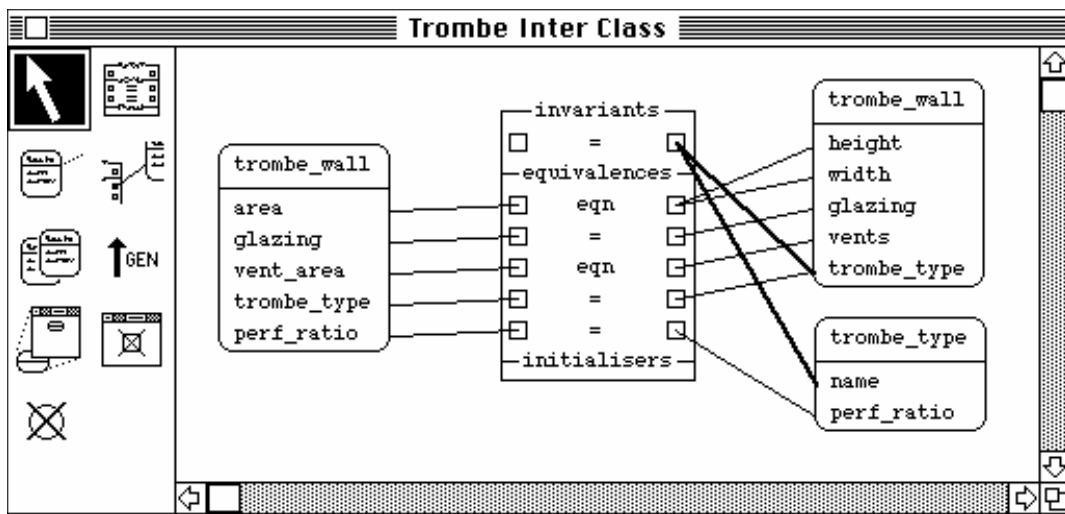
Table 1. Characteristics of the three visual mapping languages examined

Table 1 provides a summary of the characteristics of three domain-specific visual languages and their underlying metaphors that we have developed for data mapping specification. In each case, our approach is to visualise each schema’s data structure side by side, with mappings specified by visually linking between elements, in an analogous fashion to Fig. 1, but at a type rather than instance level. The “links” between elements vary in expressive ability as discussed later. In each case we have developed tools for expressing specifications and generating implementations with the ability to “reuse” the visual specifications populated with instance data to visualize mapping instances at a comparable abstraction level to the specifications. For this paper we are not concerned with surface syntax, assuming schema or instances are represented in a DOM. Our tools use several approaches to parse surface syntax into this form.

3. The View Mapping Language

The View Mapping Language (VML), and its graphical form VML-G, aim at capturing relationships between design tool views [2]. The design tools studied for this language came from the architecture, engineering and construction (AEC) domains and need to interoperate to achieve a coordinated and consistent building design. A standard approach to achieve this is to develop an Integrated Project Database (IPDB) covering the set of data requirements of the tools being integrated. With such an IPDB all interoperating tools need only map from their internal representation to the IPDB, and vice-versa, to allow consistent information flows between the tools. Conceptually this is simple. However, the IPDB for such a domain is large (the Industry Foundation Classes standard for AEC [21] has over 500 classes) and representation of similar concepts in the design tools can vary considerably. Enabling the developers of an IPDB system (professional programmers) to capture the essence of the mapping required between representations and specify detailed correspondences between them was the main goal of VML.

VML is a high-level, declarative language for describing bidirectional correspondences between two arbitrary schemas. VML eschews notions of target and source schemas in mapping definitions. As far as practicable, a VML definition treats both schemas as equal partners. This is different to most other visual mapping approaches [5, 23, 16, 27, 37]. VML also removes many distinctions between entities and attributes, to allow mappings between them to be specified in the same way as attribute-attribute mappings. A VML mapping consists of a specification of the schemas to be mapped between, and a set of correspondences between entities and attributes (called *inter_class* specifications) to describe how the mapping is to be achieved. An *inter_class* definition details: entities from the schemas that take part in the mapping; an optional set of conditions which must hold to apply the mapping (*invariants*); relationships between data in the entities (*equivalences*); and, optionally, initial values for attributes when an object is created (*initialisers*). The invariants serve two purposes in the mapping: when mapping in one direction they constrain the set of objects for which a particular mapping will be applied: in the opposite direction they provide initial values for attributes in the newly created objects.



```
inter_class([trombe_wall],[trombe_wall, trombe_type],
  invariants(trombe_wall.trombe_type = name),
  equivalences(area = height * width,
    glazing = glazing,
    vent_area = sum(vents=>(height * width)),
    trombe_type = trombe_type,
    perf_ratio = perf_ratio)
).
```

Figure 2. VML-G (top) and VML mapping for a simple correspondence between classes

VML-G uses a single icon to represent an *inter_class* definition (see centre icon in Figure 2). This has three sections corresponding to the three sections in an *inter_class* definition. These allow *invariants*, *equivalences* and *initialisers* to be grouped into localized areas in the icon and provide a visual separation of these distinct functions. The other icon type in VML-G denotes an entity taking part in the mapping with the *inter_class* (e.g. the other icons in Figure 2).

Creating a mapping within the visual environment consists of placing an *inter_class* icon between icons for the entities involved in a particular mapping. Entity icons specify the name of the entity (with optional schema and version information) and attribute and method names defined in the entity. Each individual equation, function, or procedure has a row in the *inter_class* icon. Each row end has a box to which the attributes and entities involved in the mapping element are connected. Wiring from an attribute or entity to a box connects it into that equation. Wiring an attribute or entity to a section label in an *inter_class* icon creates a new equation for that attribute or entity. Each row has a symbol defining the type of mapping being defined. These are:

- = a direct equality between an attribute (or entity) in one schema and that in the other (perhaps with type conversion). These 1:1 mappings are distinguished as they occur often.
- eqn* the attributes or entities in one schema are related to attributes or entities in the other schema through an equation (i.e. an enforceable constraint) that is not a 1:1 equivalence.
- func* the attributes or entities wired together are mapped via a functional (declarative) mapping.
- proc* a procedure maps between the specified attributes or entities. Different procedures are needed to map in each direction as in general a procedure is not automatically reversible.

Fig. 2 shows a VML-G and equivalent VML specification of a mapping between sets of classes. A 1:1 correspondence is specified between every *trombe_wall* object in one schema and every *trombe_wall* and *trombe_type* in the other schema. All attributes in one class map directly to attributes in the alternate classes (and vice-versa). There is a single invariant which matches *trombe_wall* and *trombe_type* objects with an equivalent value in the specified attributes.

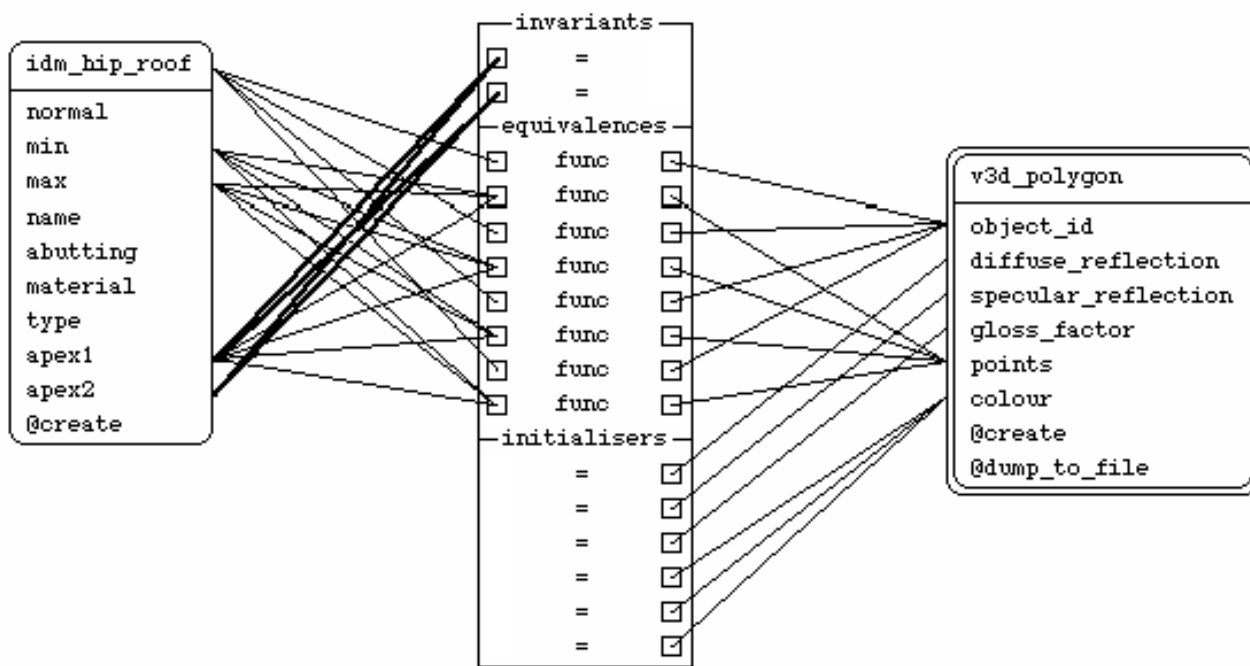


Figure 3. VML mapping with complex wiring

Fig. 3 shows a more complex mapping between each object of type *idm_hip_roof* and a collection of objects of type *v3d_polygon* (collections have a double line round the icon). In this mapping *invariants* on attributes of *idm_hip_roof* constrain when this mapping may be applied. *Initialisers* specify how to instantiate *v3d_polygon* objects. A constant number of *v3d_polygon* objects (four) are mapped from an *idm_hip_roof* object, controlled via references to objects in the set of *v3d_polygons*. It is straightforward to gain a high level understanding of the mapping from this specification and thus check its completeness (e.g., ‘Why aren’t the *normal*, *name*, *abutting*, *material* and *type* attributes mapped?’). However, mapping detail is hidden (e.g., *func* does not show the type of mapping being prescribed and *eqn* formulae are not visible). Users can open a mapping element to view and edit this textual detail. The example also shows how a mapping specification can become visually untidy if there are many overlapping links, a limitation on the efficacy of this metaphor. This can be addressed via multiple views of parts of the *same* mapping.

A complete textual mapping specification covering the entire mapping can be created for use in an IPDB. Our IPDB framework includes an interpreter for VML mappings allowing data to be mapped between the IPDB and a design tool, either by batch or incremental update of the data stores. Inconsistencies can be viewed, and appropriate action taken to ameliorate conflicts.

4. The Rimu Visual Mapper

The Rimu Visual Mapper (RVM) targets mappings between health message formats, such as those in Fig. 1 [16]. The normal approach to this problem is to develop a custom translator in a language such as C++. Such translators are very tedious to write, involving tens of pages of error prone coding by experienced programmers. Our aim in developing RVM, collaboratively with Orion Systems Ltd, was to minimise professional programmer involvement, and dramatically reduce the time and cost to develop and maintain such translators. Target RVM users are Database Administrators (DBAs) maintaining information systems that health messages are exchanged between. DBAs are familiar with data structuring and formatting issues, but have less programming experience than users of VML. Specifically, they have some understanding of procedural programming, including parameters, and familiarity with spreadsheet expression programming but not object-oriented or declarative programming expected for VML users.

Health messages consist of hierarchical record (type) structures, with repeating and optional elements. Inheritance is not used in the type definitions, nor associated behavioural code, in contrast to VML. The mapping task is thus one of data translation, however, the structural manipulations can be complex, particularly those involving repeated elements.

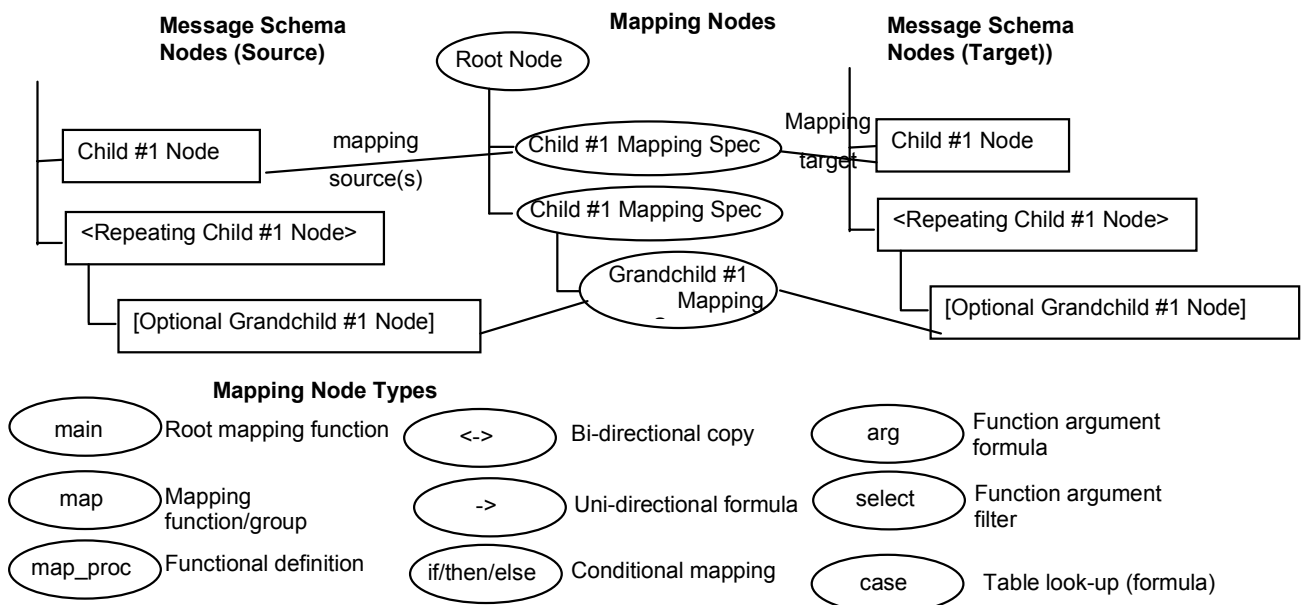


Figure 4. Basic RVM notation

Fig. 4 shows the basic elements of the RVM mapping notation, and Figs 5 and 6 show examples of its usage. The main features of the notation are:

- The visualisation metaphor chosen is a tree showing the hierarchical record structure, one element per line, indented to reflect the type hierarchy. Target end users are used to viewing message data structures in a hierarchical form, so this is a natural metaphor for them.
- Angle brackets and square braces indicate that elements are repeated or optional respectively
- Element mappings are also represented as a tree, with each element representing a mapping between one or more data elements on each side of the mapping.
- An element's mapping direction is shown by an arrow; simple equivalences are bidirectional.
- The ordering of mapping elements in the tree determines their order of application, meaning an explicit procedural sequencing of mappings, rather than VML's declarative approach.
- Levels in the mapping element tree allow for grouping of related mappings.
- If-then-else and case elements allow for conditional mappings.
- Mapping functions defined separately from the main mapping provide procedural abstraction. These take as parameters types from each side of the mapping. Function applications take

“instances” of the types as actual parameters. If a source is a collection, the mapping either selects one item and transforms it, or multiple items which it transforms one-by-one.

Figure 5 screen (1) shows two message schema (those from Fig. 1) and several mapping specifications in a prototype of RVM. The “main” mapping node (a) groups all mappings from one message to another. The first child mapping (b) groups a sequence of mappings specifying how to translate PatientMessage patient information into a PVisitMessage's fields. Node (c) specifies that MedRecNumField and IDField are (bidirectional) copies. The PIDField value is set by a default value generated by a calculation (d). The PnameField value concatenates the PatientNameRecord's LnameField and FnameField values (e). Spreadsheet-style formulae specify these unidirectional calculations. The DateRecord fields are merged to DateOfBirthField by a unidirectional function application (f). This is defined in (2); the concatenation formula is in the formula bar at the bottom, and the function can be reused to translate other dates. Another function specifies the reverse mapping (g), where DateOfBirthField is parsed to obtain the DOB.

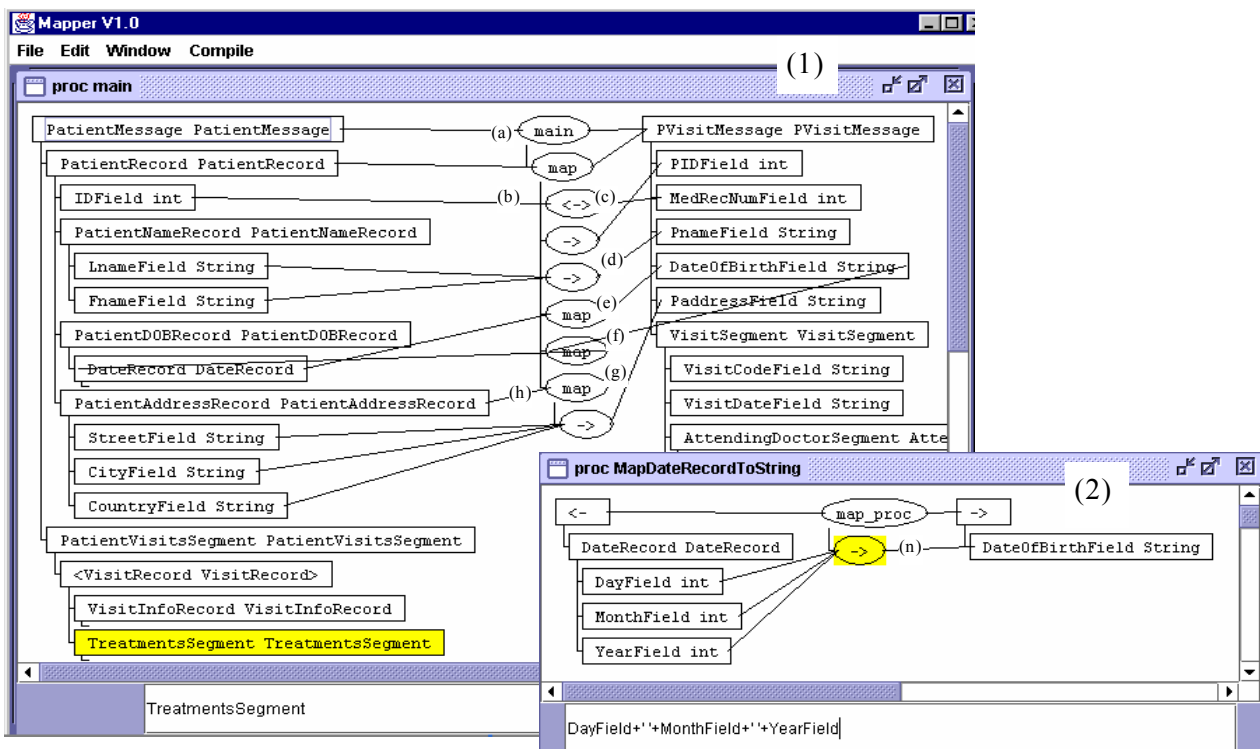


Figure 5. Example mapping specification using RVM

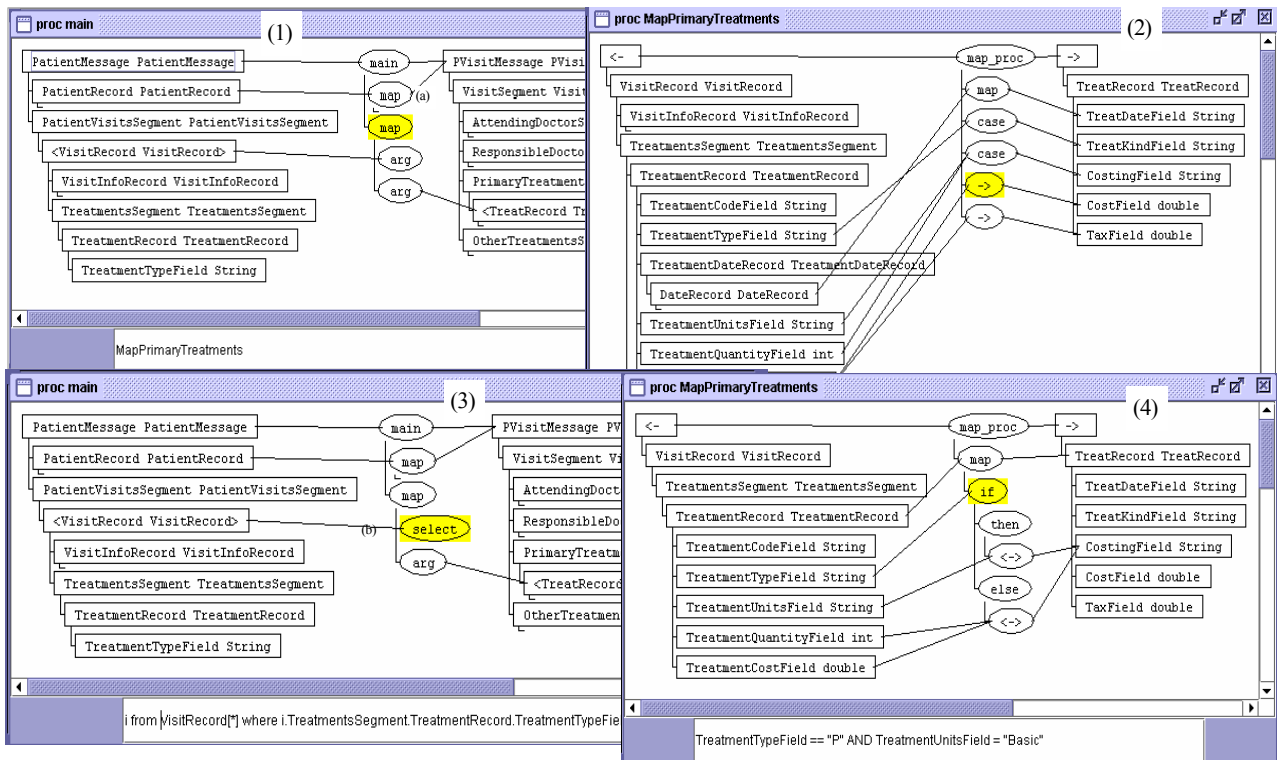


Figure 6. More complex RVM mappings

More complex mappings involve collections and conditional logic. For example, Fig. 6 screen (1) shows how PatientMessage records are translated into multiple PvisitMessage records by a mapping function applied to each VisitRecord in the PatientVisitsSegment, producing a TreatRecord in the target message's PrimaryTreatmentsSegment (a). The function definition is in (2). This uses two lookup tables to map treatment codes. Expressions on function argument nodes specify selection or filtering operations on collection arguments. For example, Fig. 6 screen (3) is an alternative to Fig. 5 screen (1), showing mapping of only "P" treatments in the source message to the PrimaryTreatments segment in the target message. The first input argument (b) now includes a selection filter over source VisitRecords, with the mapping function only applied to source records matching the criteria. Fig. 6 screen (4) shows a conditional mapping node where differing target cost information is calculated depending on a source field value.

When a mapping specification is complete, our visual mapping specification tool generates a textual "mapping language" encoding the full mapping specified by the user. This language is further compiled to a tree-structured byte code, which is interpreted by a mapping engine to automate message transformation. The message translation process is shown in Fig. 7. Mappings can be visualised as they are applied to debug them, as shown on the right of Fig. 7. This reuses the mapping specification views, annotating elements with message instance data, with navigation facilities to step through collections.

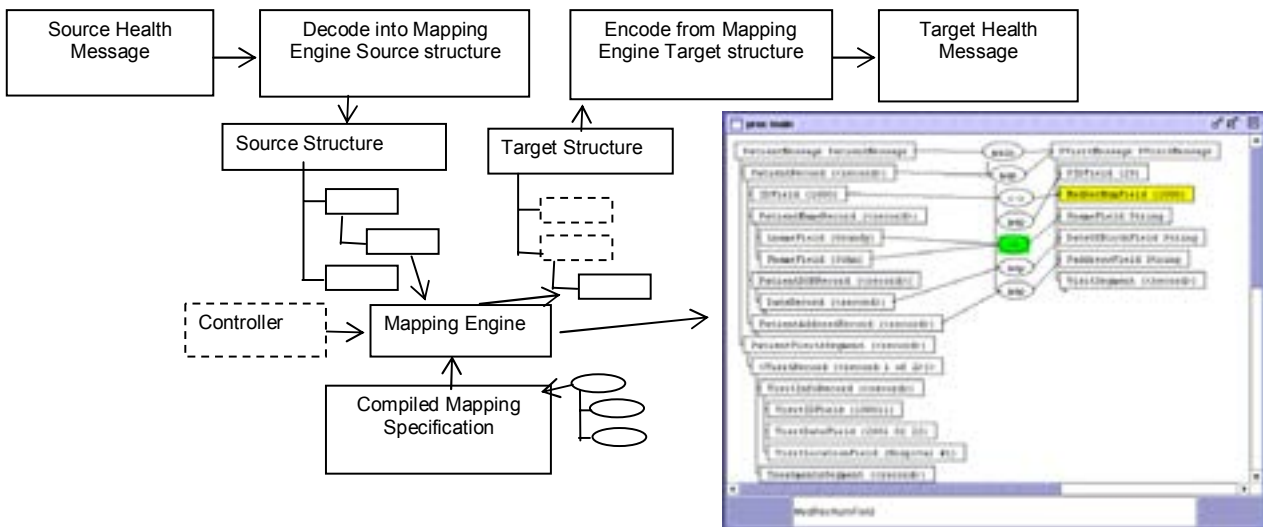


Figure 7. Message mapping engine processing

5. The Form-Based Mapper

Our final data mapping system, the Form-based Mapper (FBM) targets non programmers in the domain of business integration [26]. Most “E-business” requires data transformation from one business’s data format to another’s [4, 10, 38]. Business analysts are the professionals responsible for understanding and developing business processes; they have knowledge of what one business’s information needs are and how to map them onto another’s data. However, programmers typically implement business-to-business data exchange mechanisms.

Our aim in developing FBM [26] was to eliminate programmers from this task. Business analysts, typically have minimal programming skills; most are familiar with spreadsheet and possibly simple database programming, but little else. Accordingly we have adopted a metaphor based on business forms. Data structures are represented as business forms, and mappings are specified using a “business form copying” metaphor, with spreadsheet-style data dependency calculations. This mimics what happens when a paper form is sent from one business or unit to another and information from the form is copied into the receiving business’ data entry screens.

Fig. 8 shows the mapping specification process used by FBM. Meta-data from enterprise systems (1) is extracted to describe source and target data. A default “business form” is generated for each data model (2). A business analyst can modify these default form layouts to better conform to physical business forms (hard-copy or computer screen). The analyst specifies correspondences between fields or groups on source and target forms by direct manipulation (3). These correspondences and associated formulae are used to generate data mapping code in the form of XSLT scripts, Java programs or 3rd party data mapping tool code (4).

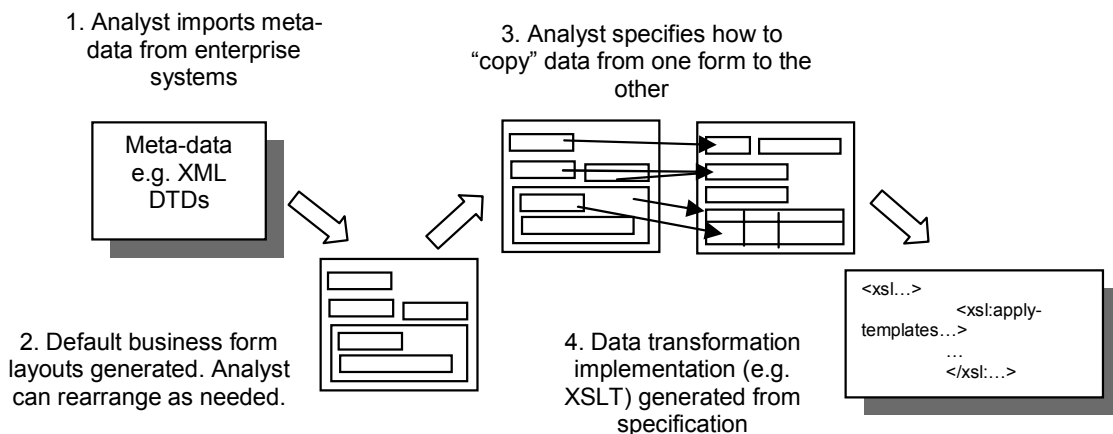


Figure 8. Business form-based data mapping specification and code generation.

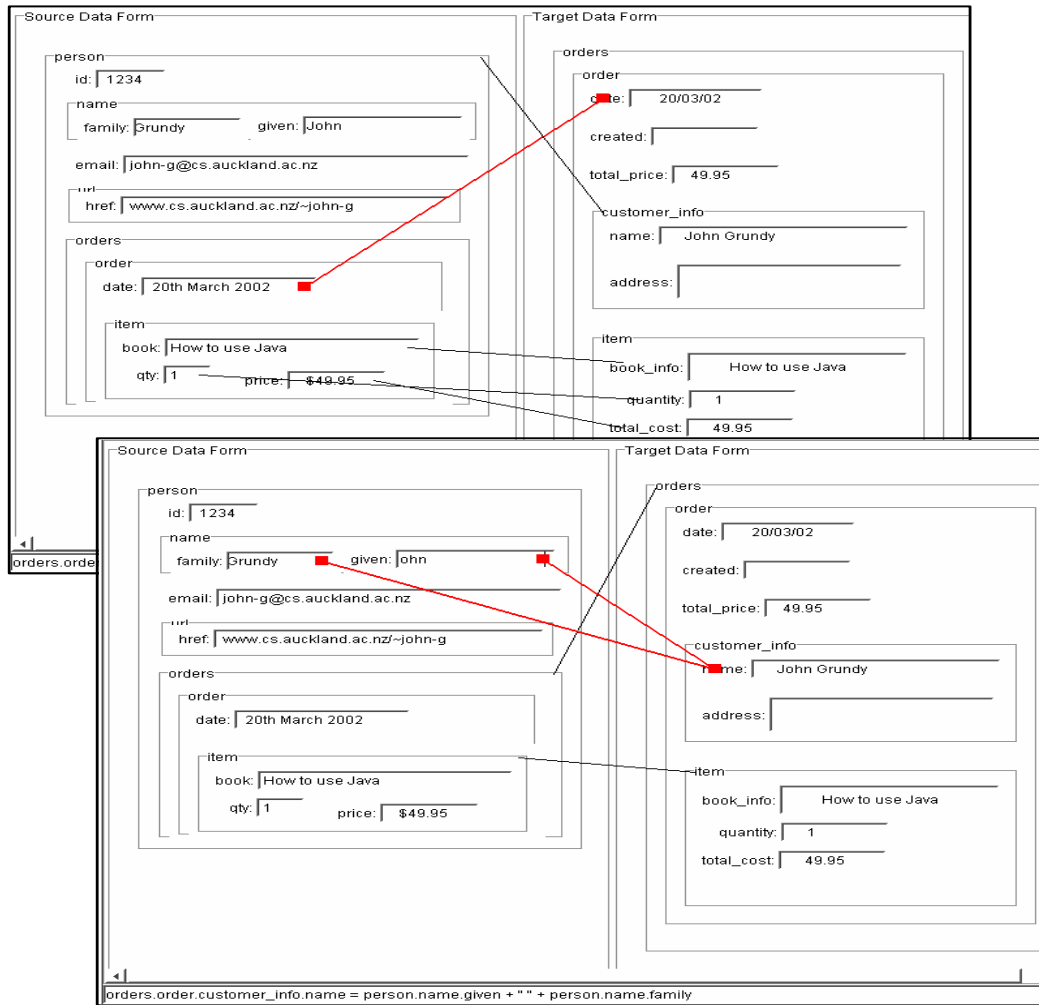


Figure 9. Simple (top) and more complex (bottom) data mapping examples.

Fig. 9 shows FBM in use, with two Order data structures viewed side by side as business forms. Unlike VML and RMV, forms are populated with actual data, to provide a more concrete metaphor. A programming-by-demonstration approach is used to specify source and target data structure correspondences. The analyst drags and drops connections between one or more fields or field groups in the source form and one or more fields or field groups in the target form to specify data transformation mappings. Fig. 9 shows the following different types of mapping:

- 1:1 copies possibly with type or name conversion, for example *qty* to *quantity*, *price* to *total cost* and *date* to *date*, the latter with a type conversion. In these cases a simple drag and drop between fields, possibly with a conversion formula, is sufficient to specify the mapping.
- 1:1 relationships between groups of fields. E.g., the 'person' record has been mapped to the 'customer_info' record.
- m:1 mappings, where several fields may be required to determine the value of one field, e.g. the mapping of 'family' and 'given' fields of the 'name' entity are mapped to the 'name' field in the target form. To specify this, the business analyst selects the two source fields and then drags to the target field, creating a 2 to 1 linkage and enters a formula to specify the calculation (in this case concatenation with a separator). As with RVM, these mappings are usually of a simpler form than their inverse, which typically require some form of parsing.
- n:m mappings, of any combination of fields and structures. For example, the order record in the source form maps onto a whole target form. This implies a single customer with multiple

orders in the source form will generate multiple target “forms” (target order data records). In this example, multiple source order items map onto the same number of target order items. Many domains involve more complex structural mappings (e.g. selection from source group to form target, m:1 or 1:m structural transformations). In FBM analysts specify correspondences between one or more source fields or groups and one or more target fields or groups. They may then qualify the correspondence indicating selection from source groups by indexing or filtering and qualify the target by specifying collection indexing or construction. In our tool the analyst uses library functions (similar to spreadsheet list/array manipulation functions) acting on source field(s) or group(s) to produce result values for the target.

6. Discussion

Here we survey related work and then compare and contrast our visual mapping languages using the well known Cognitive Dimensions framework.

6.1 Related Work

System integration has become a crucial problem in many domains, leading to the development of a wide variety of solutions [4, 38, 45]. The majority of data transformation specification techniques are “programmer-centric” focussing on supporting professional programmers. Many use custom-coded transformation modules or components that take considerable design, implementation and testing by expert programmers [12, 11, 16]. For example, most EDI solutions provide a set of predefined function libraries programmers use to encode and decode messages in particular protocols [11, 41]. Translating between message formats involves reading a message using a protocol's API, writing code to construct a new message, and then generating its transport-level representation using another protocol API. The few high-level EDI message mapping systems developed, such as ETS [1], suffer from using low-level, textual representation of mappings or overly simple visual formalisms. Message-Oriented Middleware (MoM) systems, including MQ Series™ and Tuxedo™, use a similar approach.

Many data integration products address similar problems to our VML mapper. These include Openlink Virtuoso™ [23], the Universal Translation Suite [10], and various virtual database integration [5, 27, 42, 46] and Enterprise Application Integration (EAI) tools, such as Aditel [1], eXcelon [12], Vitria BusinessWare™ [43] and BizTalk™ [11]. Some of these provide translation support for database, message and XML-encoded data using visual representations of mappings, but are limited to simple record structures and are relatively difficult for non-experts to use. Most database-oriented integration products use table-centric specification metaphors. Message-oriented products tend to adopt tree-structured visualisation metaphors, which are neither as declarative as VML nor as analyst-centric as the form-based mapper.

Message integration tools analogous to RVM include Data Transformation Manager, BizTalk [14] and MQ Integrator™ [22]. These provide limited abstract message translation facilities, typically requiring coding of complex translations. Most enable non-expert programmers and some non-programmer end users to specify message data transformations. Many adopt a tree-based mapping metaphor with drag-and-drop between tree nodes, similar to RVM. Most do not however try to capture the structure of the mappings themselves, but rather simply visualise these as links between tree nodes and leaves.

Many recent systems use scripted solutions to support easier evolution of transformations. Many use XML-based transformations [31, 39] with “standardised” DTDs and XML Schema [39, 47]. Many XML translators have been produced, including the W3C standard XML Style Sheet Transformation (XSLT) scripting language, Seeburger's data format and business logic converter [37] and eBizExchange [33]. BizTalk™ [14] and BusinessWare™ [43] also provide visual XML-based message mapping support. Most XML document translation systems use XSLT [7, 47]. These suffer from a lack of expressive power (especially for complex hierarchical mappings) and only partially support visual mapping and XSLT script generation. In addition, these tools use

programmer-centric metaphors, usually tree-based drag-and-drop with transformation expressions. These approaches do not suit many end-users, such as business analysts. XSLT provides a degree of declarative transformation support, but not to the degree of VML.

Several visual language approaches address similar themes to our work. These include SIML-based XML document representation using transformation by graph-grammars [48], algorithm animation using formal mapping specifications between structures [35], and VXT, an XML document transformer [36]. The first two use rule-based specification of mappings between structures, encoding these rules as graph transformation-based specifications or decorated abstract syntax trees. These contrast with our structure-based linking and formulae, which are more readily visualised using source/target linking approaches. VXT provides a document transformation visual language metaphor, which is quite different to our work in that it uses a visual pattern-matching approach rather than explicit structural linkages. Milicev [29] describes a system that has similar expressive power to VML but uses extended UML diagrams including conditional and iterative constructs to specify transformations.

FBM incorporates programming by demonstration (PBD). PBD systems deduce task specifications from user interactions. These are then automated or partially automated [9]. Such systems have been applied to a wide range of problem domains, including MetaMouse [30], Masuishi's report generator [28] Sugiura's Internet Scrapbook [40] and XSLByDemo [20]. A key to the success of PBD systems is the use of real-world metaphors by which users demonstrate actions and computer applications reflect learned operations to users.

6.2 Evaluation

We have undertaken usability trials of our three environments, but for the purposes of this paper it is more useful to evaluate and compare them using the Cognitive Dimensions framework [19]. The significant results of this are as follows, categorised by dimension.

Abstraction Gradient: VML has a relatively small number of visual abstractions: entity icons, inter-class definition blocks, and mapping links. There is some extra complexity in the parts of an inter class definition (invariants, equivalences, initialisers), but this is straightforward for the targeted users. This makes it easy to design and understand a mapping definition at a high level, but, hides the complexity of the formulae used to specify mapping equations and functions. These provide a large number of primitives, including difficult concepts such as bijections, pointer referencing, and escape to code for function specification.

RVM is a medium-level abstraction system. Mapping functions and groups represent potentially complex aggregates of primitive transformations and require knowledge of procedural abstraction. Collection mappings succinctly capture complex, iterative transformations without the need for iteration variables and other complex notation. Use of procedural abstraction was a deliberate choice for RVM, due to end user familiarity with the paradigm, and also as processing of messages requires explicit specification of control flow, for example to avoid multiple passes through a sequential stream of records.

The key abstraction used in FBM is the business form, a concrete visual metaphor comprising primitive form elements (labels, text fields, check boxes, etc) and groups of primitives. These abstractions map onto meta-data elements, though the user can create further abstraction groups if required. Links between fields represent formulae converting source data item(s) and group(s) to target data item(s) and group(s).

In each case, we have chosen a set of abstractions familiar to the targeted end user group. It is interesting to observe that the number of visual abstractions increases inversely to the capability of the end user. This may seem counter intuitive, but is explained by the escape to underlying richer textual formalisms available with RVM and particularly VML.

Closeness of mapping: VML's graphical notation extends from a notation that is familiar to the targeted end users, UML class icons. The inter class definitions and mapping links are intuitive in this context. There is, however, a large step from this visual form to the underlying

textual language. RVM's primitive visual elements represent simple field-level transformations in, explicitly representing source and target dependencies. Mapping functions, iterations and groupings represent high-level, aggregated dependencies of target schema items to source items using abstractions familiar to the target users. FBM uses a business form metaphor. Its visual representation thus maps directly onto business analysts' cognitive model of their problem domain. Allowing user editing of the generated form layout supports even closer mapping as analysts can tailor the layout to be closer to the actual forms they are familiar with. A major difference between the languages is that VML allows mapping of method calls from source to target, a feature omitted from FBM and RVM as a concept thought too difficult for target users.

Consistency: VML's entity and inter class nodes and RVM's schema and mapping nodes are distinguished by basic shape differences. The left-to-right connectivity of source/mapping/target nodes is preserved throughout for both. RVM's hierarchical schema and mapping node links use the same layout and visual representation. FBM's source and target form representations both use the same visual form elements. All inter-form element links are rendered consistently, presenting a potential problem as discriminating between simple and complex mappings may be desirable for the business analyst.

Diffuseness/Terseness: The VML visual notation is terse, with only three main elements, and less than a dozen minor elements. The underlying textual notation is, however, far richer, and hence more diffuse. RVM is also quite terse, using a small set of visual icons and connectors and relying on labelling to distinguish different mapping operations and abstractions. In contrast to these more abstract approaches, FBM employs a more verbose visual language that can include elements not directly used in the mapping process e.g. business form layout groups, labels, lines and boxes and images. In contrast, mapping specifications using meta-data renderings such as trees and entity-relationship diagrams seldom include elements not directly used in the meta-data mapping specification. The use of a concrete form-based metaphor in our approach necessitates a less terse notation to support the desired visual metaphor.

Visibility and Juxtaposability: All three have good juxtaposability with the ability to have multiple views open side by side, displaying different parts of the same mapping specification, and environment navigation options make it straightforward to locate related elements and views. In VML, the formulae for the mappings are specified separately and must be navigated to in separate windows. Use of elision to display the formulae within the icons would have been an alternative. In RVM, poor visibility occurs when reverse-mappings or non-hierarchical message schema references are present. Both result in source/target lines crossing over icons and other connectors, obscuring specifications. Some complex, structural mappings where formulas are used, based on source field values, to specify sub-record groups to map, can't be directly represented visually (but can be expressed in our textual mapping language and encapsulated in a mapping function node).

FBM has explicit inter-form element links providing good visibility, but the links between form elements and element groups to the underlying meta-model is hidden. When the user modifies form layout e.g. by adding or rearranging grouping, this linkage is blurred and not visible in the visual form-based visualisation nor tree-based structure views. Two views are supported in our tool: concrete form-based and tree-based structure visualisations, which are viewed side-by-side. Sub-views are supported using the tree-based view to select a portion of the form for display, but multiple views displayed simultaneously are not currently supported. The FBM views can get very large and some basic elision facilities and multiple views are provided to users. However, currently these result in loss of context of the form elements and mappings.

Viscosity: Changing mapping links in VML has low viscosity, although there is a need to be able to order mappings in the inter class icon to minimise link crossovers. Individual inter class definitions usually only involve a small number of entities, so making space for additional entities is not an issue. The hierarchical layout of RVM provides some difficulty when changes are made. These may necessitate tree manipulation operations that are currently not well

supported in the environment. FBM has a low viscosity to small changes, where the time to make a change is nearly always equivalent to the initial specification time (wiring time and specification of the mapping equation).

Hidden Dependencies: This type of dependency may occur in VML due to the declarative nature of the specification language. The order of evaluation of mappings is dependent upon the specifications which exist and not visible within the described mappings. However, as a declarative language, the fact that evaluation order changes is not significant as long as the mappings are resolvable. What may be significant is that some mappings may become resolvable with changes or additions to the mapping specification and this is not visible from the specification environment. RVM's procedural abstractions create hidden dependencies (separation of function from application), which are minimised through environment navigation aids. FBM's highly concrete metaphor minimises hidden dependencies for the user.

Error-proneness: The wiring approach of VML should reduce the chance of errors in mapping specification, but the hidden nature of the equations comprising a mapping will increase error-proneness by reducing opportunities to validate the equations, functions and procedures which are utilised for the mapping specification. Similarly, the formulae in both RVM and FBM provide significant opportunity for inadvertent errors.

6.3 Implementation techniques

VML was implemented using our MViews framework for constructing multiple view visual editing environments [15], which allowed the environment to be rapidly developed. This framework is implemented using an object oriented extension to Prolog, which was excellent for implementing the declarative features of the underlying mapping implementation.

The prototype RVM environment was implemented using JComposer [16], a meta tool resulting from further development of our MViews framework (including lessons learnt from development of VML). This allowed simple experimentation with notational features and rapid generation of the resulting environment. Additional back-end code generates textual mapping language code. The compiler and Rimu Mapping Engine are written in Java, the later consisting of a threaded interpreter for generated byte code. Orion has developed a commercial version of the mapping tool, Rhapsody [34]. This preserves the metaphor, mapping language and engine architecture, and most visual specification and visualisation techniques. Modifications include phasing out conditional nodes, better navigation, better support for non-hierarchical field references and additional visual annotations.

FBM was developed using Java's Swing GUI API and JAX XML parsing API, and hence required more programming effort than VML or RVM. The tool allows users to import meta-data from XML-encoded data files or DTDs. The meta-data is used to generate a basic form layout with simple heuristics used to generate form elements and element groupings using Java Swing components. We used the Java Swing component event-passing mechanism and transparent panel overlays to intercept user interaction with the generated form components for drag and drop. We generate XSLT-based transformation scripts to implement the data mapping specifications.

7. Conclusions and Future Research

We have presented three domain-specific visual languages addressing the problem of specifying mappings between different schema. The mappings differ in their intended target user group, and hence have adopted different data visualisation metaphors. Each metaphor builds from concepts familiar to the specific target group, and closely related to the business problem they are trying to solve: OO design diagrams for programmers; tree-based schema representations for DBAs; and business forms for business analysts.

The resulting Visual Languages are quite different in nature, yet have surprisingly similar expressive ability for the relatively simple XML-based structures. An upcoming piece of work will attempt to characterise more formally the expressive ability of visual languages for mapping

to provide a deeper understanding of what is forsaken by various choices of formalism. Our three visual languages are unified by the use of a wiring metaphor to express mapping relationships. This appears to us to be a natural way to represent such relationships, but has the problem of “clutter” for large mapping specifications, which must be solved by multiple views or elision.

In each case, the visual language hides much of the complexity of the mapping, in particular the manipulation of collections. This has the advantage that it allows users to concentrate on the high level design of the mappings, i.e., what elements are related to one another, and understand at a gross level the structural manipulations involved in collection mapping. However, it means that complex mappings can be difficult to depict or comprehend from the visual representation, requiring careful examination of formula boxes and textual renderings to understand and debug. We are skeptical, however, that provision of convincing visualisations of complex collection manipulations is achievable for the two less programmerable target groups. For skilled programmers, an underlying visual formalism, such as a Prograph-style dataflow language [8], may prove an option, but it is questionable whether this has a significant advantage over the textual code for such a target group.

There is an interesting tradeoff with these visual notations between having efficient representations of schema in order to describe a mapping and utilising standard representations of schema that will be immediately comprehensible to users. For example, EXPRESS-G [24], used in engineering domains to represent schema, is poorly suited to the overlay of a mapping notation, yet the introduction of a new representation for schema in this domain purely to support mapping specifications will meet with resistance.

It is clear that the support environment offered alongside the visual notation has a major impact on its usability. The support for data and formula views in FBM is an important aspect of its usability, as is the partial mapping evaluation functionality to view exemplars of transformed data. Multiple overlapping views of partial mappings with consistency maintained across them provide some controls for managing large schema with the notations. Further environment support for the visual languages comprising libraries of common mappings and automated matching of schema portions may also be an important adjunct to the visual notation.

We are interested in exploring other metaphors for the mapping problem. In current work, we are examining approaches to specifying the mapping of visual notations from one to another, together with approaches for specifying office automation (with attendant data mappings).

References

1. Aditel. <http://www.aditel.org>
2. R. Amor, A Generalised Framework for the Design and Construction of Integrated Design Systems, *Ph.D. thesis*, Department of Computer Science, University of Auckland, Auckland, New Zealand, 1997, 350pp.
3. R. Amor, J. Hosking and W. Mugridge, ICAtect-II: A Framework for the Integration of Building Design Tools, *Automation in Construction*, 8(3), 1999, 277-289.
4. G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt and N. Weiler, WISE: business to business e-commerce. Proceedings Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises, RIDE-VE'99, Los Alamitos, CA, USA. IEEE CS Press, 1999, 132-139.
5. C. Batini, M. Lenzerini and S.B. Navathe, A Comparative Analysis of Methodologies for Database Schema Integration, *ACM Computing Surveys*, 18(4), December, 1986, 323-364.
6. J. Blackham, P. Grundeman, J.C. Grundy, J.G. Hosking and W.B. Mugridge, Supporting Pervasive Business via Virtual Database Aggregation, In Proceedings of Evolve'2001 – Pervasive Business, Sydney, Australia, March 15-16, DSTC Press, 2001.
7. D. Cheung, S.D. Lee, T. Lee, W. Song and C.J. Tan, Distributed and scalable XML document processing architecture for E-commerce systems. *Proceedings of the Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*. IEEE, 2000, 152-157.

8. P.T. Cox, F.R. Giles and T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, in *Proceedings of the 1989 IEEE Workshop on Visual Languages*, IEEE CS Press, 1989, 150-156.
9. A. Cypher, (ed) *Watch what I do: Programming by demonstration*, Cambridge, Mass, MIT Press, 1993.
10. Data Junction Corp, <http://www.datajunction.com/>.
11. M.A. Emmelhainz, *Electronic Data Interchange: A Total Management Guide*, Van Nostrand Reinhold, 1990.
12. eXcelon Corp, eXcelon B2B Integration Server White Paper, <http://www.exceloncorp.com/>.
13. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh, Inconsistency Handling in Multiperspective Specifications, *IEEE Transactions on Software Engineering*, vol. 2, no. 8, 1994, 569-578.
14. M.A. Goulde, Microsoft's BizTalk Framework adds messaging to XML, *E-Business Strategies & Solutions*, Sept. 1999, 10-14.
15. J.C. Grundy, J.G. Hosking and W.B. Mugridge, Inconsistency Management for Multiple-View Software Development Environments, *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, November, IEEE CS Press, 1998, 960-981.
16. J.C. Grundy, W.B. Mugridge, J.G. Hosking and P. Kendal, Generating EDI Message Translations from Visual Specifications, In *Proceedings of the 2001 IEEE Automated Software Engineering Conference*, San Diego, 26-29 Nov, IEEE CS Press, 2001, 35-42.
17. J.C. Grundy and W. Zou, Building multi-device, adaptive thin-client web user interfaces with Extended Java Server Pages, accepted as a chapter in *Cross-Platform and Multi-device User Interfaces*, Wiley, 2003.
18. J.C. Grundy, J. Bai, J. Blackham, J.G. Hosking and R. Amor, An Architecture for Efficient, Flexible Enterprise System Integration, In *Proceedings of the 2003 International Conference on Internet Computing*, Las Vegas, June 23-26, CSREA Press, 2003.
19. T.R. Green and M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing*, (7), 1996, 131-174.
20. M. Hori, T. Koyanagi, K. Ono, M. Abe, *XSLByDemo*, www.alphaworks.ibm.com/tech/xslbydemo, 2000.
21. IAI, International Alliance for Interoperability <http://www.iai-international.org/>, 2003.
22. IBM Corp, *MQ Series Integrator*, <http://www.ibm.com/>.
23. K. Idenhen, Introducing OpenLink Virtuoso: Universal Data Access Without Boundaries, White paper, <http://www.openlinksw.com/>.
24. ISO/TC184, Part 11: The EXPRESS Language Reference Manual in Industrial automation systems and integration - Product data representation and exchange', *International Standard, ISO-IEC, Geneva, Switzerland, ISO 10303-11*, 1992.
25. W. Kim and J. Seo, Classifying Schematic and Data Heterogeneity in Multidatabase Systems, *IEEE Computer*, 24(12), December, 1991, 12-18.
26. Y. Li, J.C. Grundy, R. Amor and J.G. Hosking, A data mapping specification environment using a concrete business form-based metaphor, In *Proceedings of the 2002 International Conference on Human-Centric Computing*, IEEE CS Press, 2002, 158-167.
27. E.P. Lim and R.H.L. Chiang, The integration of relationship instances from heterogeneous databases. *Decision Support Systems*, vol.29, no.2, Aug, Publisher: Elsevier, Netherlands, 2000, 153-167.
28. T. Masuishi and N. Takahashi, A Reporting Tool Using Programming by Example for Format Designation, in Liebermann H (ed) *Your Wish is my Command*, Morgan Kaufman, 2000.
29. D. Milicev, Automatic model transformations using extended UML object diagrams in modeling environments, *IEEE ToSE*, 28(4), 2002, 413-431.
30. D. Maulsby and I.H. Witten, MetaMouse: an instructable agent for programming by demonstration, in Cypher A (ed) *Watch what I do*, MIT Press, 1993.
31. J.P. Morgenthal, XML: The New Integration Frontier, *EAI Journal*, Feb., <http://www.eaijournal.com/>, 2001.
32. B. Nuseibeh, Towards a framework for managing inconsistency between multiple views, in *Proceedings of Viewpoints'96*, ACM Press, San Francisco, 1996, 184-186.
33. OnDisplay Corp, <http://www.ondisplay.com/>.
34. Orion Systems Ltd, Rhapsody Integration Engine, http://www.orion.co.nz/rhapsody_overview.htm.

35. M. Pereira and P. Henriques, Visualisation/animation of programs based on abstract representations and formal mappings, In Proceedings of the 2001 IEEE Human-Centric Computing Conference, Sept 5-7, Stresa, Italy, IEEE CS Press, 2001, 373-380.
36. E. Pietriga and V. Zumer, VXT: Visual XML Transformor, Proceedings of the 2001 IEEE Human-Centric Computing Conference, Sept 5-7, Stresa, Italy, IEEE CS Press, 2001, 404-405.
37. Seeburger Corp, <http://www.seeburger.de/xml/>.
38. J.A. Senn, The evolution of business-to-business commerce models: the influence of new information technology models. Proceedings of International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems, Piscataway, NJ, USA, IEEE CS Press, 1999, 153-158.
39. H. Spencer, XML standards for data interchange. *Imaging & Document Solutions*, vol.9, no.9, Sept. 2000, 15-17.
40. A. Sugiura, Web Browsing by Example, in Liebermann H (ed) *Your Wish is my Command*, Morgan Kaufman, 2000.
41. P.M.C. Swatman, P.A. Swatman and D.C. Fowler, A model of EDI integration and strategic business reengineering. *Journal of Strategic Information Systems*, vol.3, no.1, March, 1994, 41-60.
42. E.M.A. Uchoa and R.N. Melo, HEROS: a framework for heterogeneous database systems integration. Database and Expert Systems Applications. 10th International Conference, DEXA'99, Lecture Notes in Computer Science Vol.1677, Berlin, Germany, Springer-Verlag. 1999, 656-667.
43. Vitria Technolgy Inc, <http://www.vitria.com/>.
44. P. White and J.C. Grundy, Experiences Developing a Collaborative Travel Planning Application with .NET Web Services, In Proceedings of the 2003 International Conference on Web Services, Las Vegas, June 23-26, CSREA Press, 2003.
45. H. Wing, R.M. Colomb and G. Mineau, Using CG formal contexts to support business system interoperation. In *Proceedings of the 6th International Conference on Conceptual Structures*, Berlin, Springer-Verlag, 1998, 431-438.
46. E. Wu, A CORBA-based architecture for integrating distributed and heterogeneous databases, Proceedings Fifth IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, CA, USA, IEEE CS Press, 1999, 143-152.
47. XML.org, *XML and XSLT*, <http://www.xml.org/>.
48. K. Zhang, D.-Q. Zhang and Y. Deng, A visual approach to XML document Design and Transformation, In Proceedings of the 2001 IEEE Human-Centric Computing Conference, Sept 5-7, Stresa, Italy, IEEE CS Press, 2001, 312-319.

7.2 A domain-specific visual language for report writing

Dantra, R., Grundy, J.C. and Hosking, J.G. A domain-specific visual language for report writing, In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, Cornwallis, Oregon, USA, Sept 20-24 2009, IEEE CS Press, pp 15-22.

DOI: [10.1109/VLHCC.2009.5295308](https://doi.org/10.1109/VLHCC.2009.5295308)

Abstract: Many domain specific textual languages have been developed for generating complex reports. These are challenging for novice users to learn, understand and use. We describe our work developing the prototype of a new visual language tool for a company to augment their textual report writing language. We describe key motivations for our visual language tool solution, its architecture, design and development using Microsoft DSL tools, and its evaluation by end-users.

My contribution: Co-developed key ideas for this research, co-designed approach, co-supervised Masters student, wrote substantial parts of paper, co-lead investigator for funding for this project from FRST and Technology NZ

A Domain-Specific Visual Language for Report Writing Using Microsoft DSL Tools

Ruskin Dantra¹, John Grundy² and John Hosking³
Prism New Zealand¹, Dept. of Electrical and Computer Engineering² and Dept. of Computer Science³
University of Auckland
Auckland, New Zealand
ruskin.dantra@gmail.com¹, {john-g², john³}@cs.auckland.ac.nz

Abstract

Many domain specific textual languages have been developed for generating complex reports. These are challenging for novice users to learn, understand and use. We describe our work developing the prototype of a new visual language tool for a company to augment their textual report writing language. We describe key motivations for our visual language tool solution, its architecture, design and development using Microsoft DSL tools, and its evaluation by end-users.

1. Introduction

Enterprise tasks and processes tend to be complicated and require considerable end-user training and support. Writing reporting scripts to query databases and display the extracted information is a good example of this. Industrial databases are complex and hold large amounts of data, the majority of which is irrelevant to the user. They also have unfriendly features such as arcane table naming, implicit relationships and cascading associations [1].

While there are many existing report writer approaches [2] [1] [3] [4], these are typically low level in their approach. Generic reporting tools often do not cater for the exact needs of an end-user for a particular enterprise database. This forces an enterprise (organization) to introduce their own proprietary textual domain specific languages (DSLs) which require significant programming ability on behalf of the report designers. Issues in using such languages include arcane syntax, hidden dependencies and a lack of support tools making the languages difficult for novice users thus decreasing their productively [1] [5] [6].

This paper describes the development of a visual domain specific visual language (DSVL) and an environment that assists end-users to rapidly design and implement reports sourced from an enterprise system and its database for the printing and graphics art industry. This is done by creating a meta-modeling

framework which also in turn assists developers to modify and add new elements to this reporting language.

The language and environment replaces an existing textual domain specific language providing a more accessible and productive approach to report writing for novice and intermediate end-users. Our contributions are two-fold: the design, prototyping and evaluation of a visual report writer language and support tool; and as an example of industrial usage of a domain-specific visual language and meta-tool development approach.

We begin by describing the domain of the target enterprise system and its existing textual report writing language. During the description we will also highlight its inherent issues which motivated our new approach. This is followed by a description of our new DSVL, environment and framework. Following this is case study of its use to clearly demonstrate the improvements a given DSVL has over its textual counterpart.

We then describe the details of the DSVL implementation, using Microsoft DSL Tools [7]. We then describe the evaluation methods and present its results. The paper concludes with a brief set of conclusions and possible future work.

2. Background and Motivation

Prism WIN is a fully integrated and configurable enterprise Management Information System (MIS) for the printing and graphics arts industry [8]. Prism WIN has a number of integrated modules for estimation, inventory management, production management, financial reconciliation, sales ordering and processing and facilities management. The Prism report writer language (RWL) is an interpreted textual DSL that allows Prism WIN end-users to write powerful data mining queries and present the results graphically as reports [9].

Figure 1 shows a textual RWL script for a report that extracts the customers from a Prism database and

prints out their active jobs, i.e. those that are beyond quoting.

```

1. Code CASE_STUDY_1
2. Type Standard
3. Access STSR
4.
5. Scan RM
6.     Print RM_CUST + RM_NAME;
7.     Print "All Jobs For " + RM_NAME;
8.     Scan QM
9.         Choose (QM_CUST_CODE, MATCH, RM_CUST)
10.        Choose(QM_QUOTE_JOB, MATCH, QMM_JOB)
11.
12.        Print QM_JOB_NUM + QM_TITLE;
13.     End
14. End
15. Print StandarReportFooter;

```

Figure 1. RWL script example

Lines 1 to 3 are the header definition (*Controlline*), which provide metadata for searching and indexing this particular report through the Prism WIN MIS. Line 5 is a *Scan*. This loops through the specified Prism WIN database view (*RM*), printing out customer information. This is followed by an inner *Scan* which loops through the QM database view selecting active jobs for the current customer in the outer *Scan* and prints out information for each customer-job record. A *Scan* can be simply seen as a *Select* statement within the Structured Query Language (SQL) and the *Choose* statements allow two *Scans* to be joined (inner join). For this example the first *Choose* joins the *RM* and *QM* views and the second *Choose* only selects *Jobs* (not quotes).

The *Scan* is followed by a standard footer which simply prints any arbitrary (configurable) text after the end of the report.

RWL also has variables, including clumps which are aggregates of arbitrary objects (such as database columns and/or other variables), report headers and footers and many inbuilt functions. Prism has also developed an IDE for RWL, Prism Scribe.

This provides wizard support for common templates, libraries of metadata, functions and code snippets, syntax colouring and completion. This IDE is, however, all hard coded in VB6 making syntax and environment changes difficult.

As the current IDE was incompatible with Microsoft Vista and changes were also desired to extend RWL's functionality, Prism was motivated to explore alternative approaches to report specification. In particular, Prism wished to explore a more model driven approach, based on a comprehensive meta-model for RWL specification and a compatible DSVL and environment for model driven report specification.

Therefore the key enterprise requirements arising are to allow end-users to query their complex database schema to mine information and to allow the end-user

to lay this information out as they need. Secondary requirement was also to allow developers to roll out new reporting features with ease. These enterprise requirements were translated to the following technical requirements:

- An extensible meta-model that correctly represents an abstraction of RWL and its semantics
- An extensible DSVL and environment that allows end-users to easily specify and test Prism reports. This should use an end-user accessible metaphor, validate RWL semantics and constraints, provide simple access to required database meta-data, support versioning and test cases and afford modularity and scalability
- A code generator that generates RWL reports from DSVL specifications
- Delivery using a widely used platform, such as Microsoft Visual Studio or Eclipse

Many report writing tools have been developed. These include Crystal Reports [10], Visual FoxPro report writer [3], Eclipse business intelligence tools, and Visual Studio report designer [11]. These approaches use a combination of wizard-driven templates and/or textual domain-specific visual languages. Some, such as Crystal reports, provide a limited visual report layout designer and wizards to set up data sources. However most are generally designed for programmers or experienced data modellers and being general-purpose are not Prism-specific tools. Moreover Prism RWL offers integrated support with the Prism WIN MIS which other report writing languages may offer with a strenuous API or not offer one at all.

A range of other approaches have been developed to provide more end-user friendly report writing tools. FastReport Studio [12] provides visual layout and visual query specification for any ODBC source. As mentioned earlier this is also another general purpose solution and not specific to Prism. Customization is likely to be difficult and the software itself is intended for end-users with some programming knowledge. Report Sharp Shooter [13] provides a visual layout designer, report wizard, reusable templates with cascading structures, an accessible DOM-based model, preview and export. It is also intended for programmers.

FoxQ [5] uses a form-based approach to specifying XQuery XML document queries, allowing high-level authoring of reports. This is intended for document processing rather than database query and reporting [5].

A number of visual query languages have been developed, some with reporting capabilities. GQL [14]

provides a general purpose graphical query language, though not oriented to business reporting.

A WebMLbased visual language tool [15] provides GIS-oriented database querying and reporting though not business oriented reporting. HyperFlow [16] provides general purpose end-user oriented information analysis support with a visual query metaphor.

IViz uses a dataflow visual language to synthesize complex graphic reports from data [17]. Co-ordinated Queries [4] and web server page generators [6] provide visual abstractions for specifying both queries and contents of reports. Neither is oriented towards business reporting per se and neither to Prism specific databases.

3. Our Approach

We determined that we would need to design a domain-specific visual reporting language for Prism and build a prototype tool to support it. The extensibility requirements, however, meant that we had two target end-user groups to consider: Prism developers who would maintain and extend the meta-model, DSVL and environment; and end-user report authors. Our approach involved four core steps:

1. Designing a visual RWL meta-model, providing a representation of the core modelling elements and inter-relationships of the textual RWL
2. Specifying RWL constraints on the meta-model: this provides the main semantics of the language
3. Exposing the meta-model to report authors via a visual language: this involved designing a suitable visual language oriented towards end-user report writer authoring and building a visual modelling tool using this language to populate instances of the meta-model while adhering to its constraints
4. Designing and deploying code generators to generate textual RWL script from the visual RWL model: we decided to generate textual RWL scripts from our DSVL so that we could leverage the existing complex Prism report generation engine.

Several approaches were possible: (a) a WYSIWYG report designer that allows end-users to manipulate something close to the final appearance of their report, like MS Word for documents; (b) an abstract representation of report layout e.g. broken into headers/body/footers; (c) a visual representation of Prism textual report script elements but making explicit their relationships; or (d) a dataflow-like language describing data sourcing, transformation and output.

As mentioned before all of these approaches have trade-offs in terms of ease of use, closeness of mapping to the problem domain, expressiveness, scalability and ease of implementation.

Conceptually a WYSIWYG approach is appealing [5]. However, as reports have complex repeating groups, database queries and formulaic translations of data for output, providing such an approach limits expressiveness and is very challenging to implement.

Abstract report layout e.g. as header/footer/body/group bands, is quite common [11] [3] [13]. However, this approach suffers from introducing many hidden dependencies and can be very “viscous” i.e. hard for end-users to change report designs. The dataflow metaphor appears attractive and has been used in a number of visualization and/or reporting-oriented approaches [17] [18]. However these rapidly become very complex as queries and transformations grow. After surveying target end-users with mock-up design examples we chose to use option (c) above: developing a new visual language based on existing Prism textual report scripting language constructs, analogous to [14] [4]. This provides existing reporting abstractions with most dependencies made explicit via relationships; grouping of hierarchical constructs; and database element and report element constructs explicitly linked. We also borrowed the “banding” concepts from approach (b) but use the “swim lane” metaphor from activity diagrams to realize this. We found this easier and more convenient for end-users than rigid report bands and layout.

We tried two variations of approach (c), which are shown in Figure 2 and Figure 3 below.

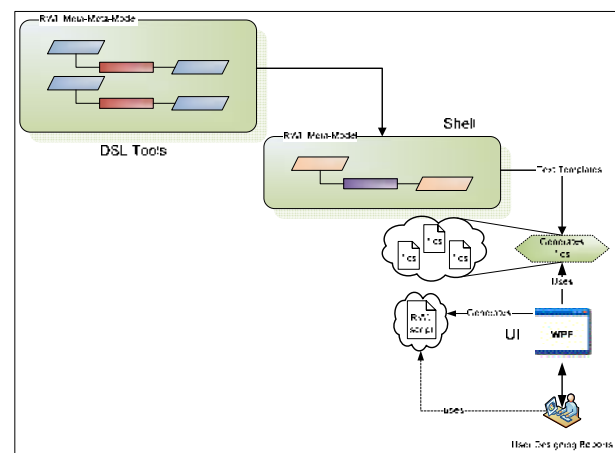


Figure 2. Approach one

Figure 3 shows an outline of the approach we took. The approach shown in Figure 2 was discarded in the early design stages due to the overhead of

implementing a whole new user interface to accentuate the RWL meta-model as Microsoft DSL Tools already offered an integrated shell to host visual languages. A trade-off had to be made between flexibility and a rich user interface experience which was offered by our first approach against rapid prototyping which was offered by the second.

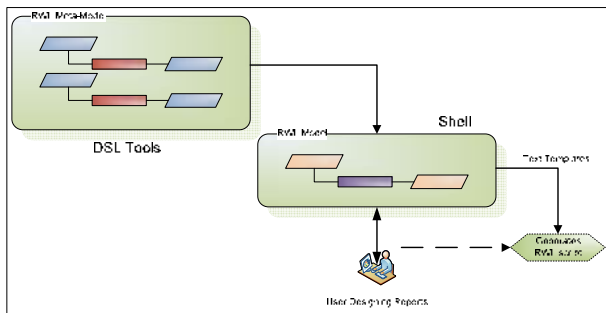


Figure 3. Approach two

Therefore our approach (shown in Figure 3) comprised of three core phases, each phase mapped to one or more steps enlisted earlier in this section.

1. Design the meta-model of the RWL representing as much information as possible (mapped to Step 1)
2. Constraints which cannot be represented using the meta-model were encoded within a rules engine via code (mapped to Step 2) [9]
E.g. Mandatory fields within a meta-model element;
Scan must be mapped to a database view
3. Developed a formal visual notation and tool for our meta-model which can be used by end-users. The tool will be used to generate the required RWL script from the corresponding visual RWL model (mapped to Step 3 and 4)

4. Usage Example

We briefly illustrate end-user report specification using our visual RWL tool. Initially the user creates a new report or opens an existing report. Figure 4 shows the end-user interface while beginning design of a new Prism report. A palette (1) provides available report elements, including header, body, footer, variables, control lines, scans, pages, comments, literals, functions, clumps, etc. We chose to use a set of “swim lanes” (2) to group report elements into key sections including Header, Body, Footer, and Variables, note

that the orientation of the swim lanes can be easily altered if needed. A property sheet (3) provides context-sensitive element properties for editing. For example, the properties for the *ControlLine1* element (4) are shown.

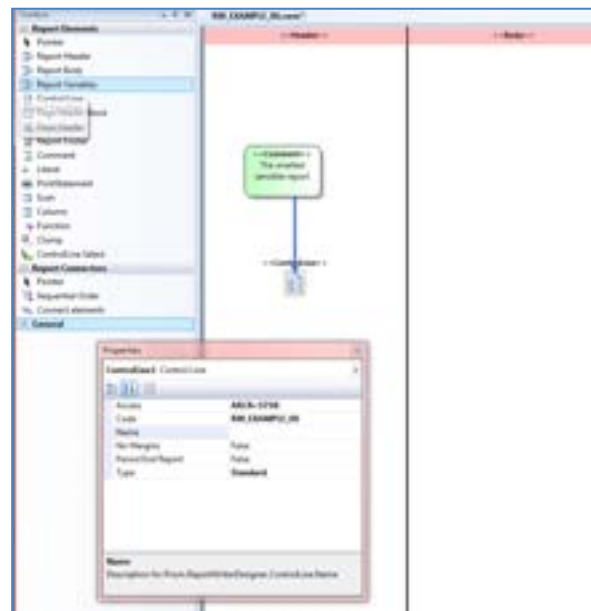


Figure 4. Visual report writer tool user interface

In this example the user has added a header and body element by drag-and-drop from the tool palette, an initial *ControlLine* to the header and initial *Scan* to the body of the report. Properties may be set when an element is added or later when selected by the user.

One of the more complex report elements supported by the Prism report writer language is the “*Scan*”, or query from a Prism database table. In Figure 6 a *Scan* element is added by the user to query a Prism database table for data items to add to the report. The user needs drags-and-drops the *Scan* from the palette (Figure 4 (1)) onto the canvas and then selects the table on which to perform the *Scan*. This can be followed by adding any conditional statements (*Choose*) which can be done by a context menu (Figure 6 left) followed by selecting the conditions (Figure 6 right). Other report elements can be added using a similar approach of drag-and-drop; link to existing elements; or specify properties. The tool enforces underlying RWL meta-model semantic constraints e.g. you cannot add clump outside a scan; you cannot add a scan to a header; and so on.



Figure 5. Design evolution while designing the report



Figure 6. Adding and specifying a Scan

The user continues to add report elements and evolve the design: e.g. report footer; page header/footer; print statements, further scans; scan elements; literals and so on.

For example, in Figure 5 (left) the report has been extended to include two print statements (that generate report body content); *Scans* (selections) from the Prism RM and QM tables (middle); table field items to appear on the printed form. Sequencing can be specified for report items to enforce ordering of printed items on the final report (Figure 5 (left), arrowhead going from *ControlLine* to *Scan*). Figure 5 (middle) also shows how two Scans can be automatically joined on their mapping column automatically. This join is configurable by the user after it is added; Figure 5 (right).

To manage complexity of reports elision and additional visual report writer diagrams are supported. The user can collapse/expand items and their sub-items on the report. They can also create additional diagrams to specify parts of the report and package these via parameterization for reuse. Figure 5 (middle) (and Figure 6 (left)) shows an example of a collapsed *Print* statement. The user can generate a textual Prism report script from the visual report writer language. Figure 7 shows the Prism report script generated from the example in Figure 6. This report script can then be run by Prism report engine against the Prism ERP database

to generate a report to a file. The resulting report file is then opened and report content shown to the user.

```

1: .....
2: <<meta-generation>>
3: <<
4: << This code was generated by a tool.
5: <<
6: << Changes to this file may cause incorrect behavior and will be lost if
7: << the code is regenerated.
8: <<
9: << Generated on 8/20/2000 1:25:26 p.m.
10: <<
11: <<-----
12: <<<meta-generation>>
13: <<
14: << Table: RM_Inventory_20
15: << Type: Standard
16: << Access: RW
17: << Name: Report in User Example 01
18: <<
19: <<
20: << Scan 01
21: << From: RM_Inventory_20
22: << Scan 01
23: << From: QM_Inventory_2000, Name: RM_Inventory_20
24: <<
25: << Print 01,Title:
26: <<
27: <<
28: <<
29: <<
30: >>>

```

Figure 7. Generated textual RWL report

5. Architecture and Implementation

Prism's requirements motivated a model-driven engineering approach to the visual tool development allowing us to capture the existing textual RWL concepts and rapidly prototype a tool [19] [20]. We chose to use the Microsoft Domain-Specific Language (DSL) tools platform to design and build our prototype [21] [7]. Other meta-tool platforms, such as MetaEdit+ [22], Eclipse GMF and our own Marama Eclipse-based platform [23] were possible choices. We chose to use Microsoft DSL tools as Prism desired a Microsoft-based solution and we believed it provided a suitable platform for robust visual report writer delivery to end-users. To realise our visual report writer prototype we followed the process outlined in Section 3:

- Meta-model development: we built and refactored over time a large meta-model to capture all of the commonly-used Prism textual reporting language elements and many of the more obscure ones. Part of this meta-model is shown in Figure 8. We designed our meta-model to be as extensible as possible; one of the major attractions of the DSL tools approach is that Prism developers can

extend this meta-model to add new visual report writer features easily.

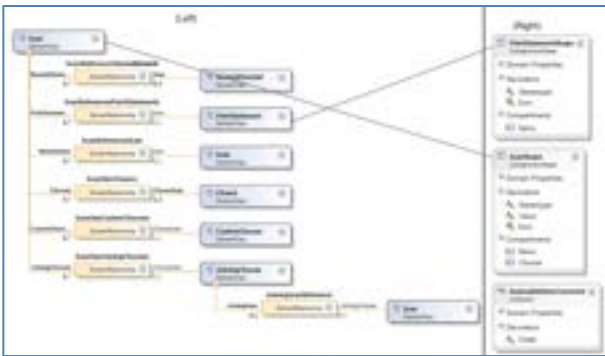


Figure 8. RWL meta-model

- Specify constraints on the meta-model: we used the DSL tools constraint specification facilities to constrain the semantics of our visual reporting tool. We also used where necessary C# functions and custom property sheets and choosers. These allowed us to implement e.g. meta-data querying facilities to the existing Prism database. This ensures end-users only use valid tables, fields and types in their reports.
- Visual report writing language: we used the DSL tools facilities to specify visual forms of meta-model elements and relationships i.e. icons and connectors to represent meta-model elements in the visual report writer diagrams. Each meta-model element has a visual form, as shown by the examples in Figure 8 (right). We used DSL tools visual editor constraints to provide layout capabilities including swim lanes, collapsing and expanding, and automatic layout of sub-items for nested elements such as the *Print* and *Scan* model elements.
- Code generator: we wanted to use the existing Prism reporting engine to actually interpret and run reports specified in our visual report writing language. The easiest way to achieve this was to generate textual report script and run it via the Prism reporting engine. We used the T4 templating engine within the DSL tools to synthesise textual report scripts from the report model and show the user the result which can then be importing and run via the Prism MIS.

6. Evaluation

We undertook two complementary evaluations of our solution. The first was a continuous design evaluation. This used the Cognitive Dimensions (CD) framework [23] during each design iteration of the

DSVL and environment to evaluate the tradeoffs made and inform the next iteration. The second was a pair of end-user evaluations using a combination of tool familiarization and a survey instrument.

The CD evaluation at the final iteration gave the following results. The underlying “inner join” (*Choose* statements) based nature of report specification requirements naturally leads to *hard mental operations*, particularly for the non-programmer. We mitigated these by using layout (including automatic layout) in a first class way (rather than as *secondary notation*), annotation mechanisms to emphasize both the sequential and hierarchical nature of some aspects of the specification and context sensitive helpers. This approach emphasizes the report *logic* over the final report *layout* (more conventional report designers emphasize the latter) which is a tradeoff of concreteness of the final report versus a better *closeness of mapping* to the information access requirements for the data constituting the report. The latter was chosen to emphasize as Prism felt this was the area where most report specification errors occurred. The auto layout mechanisms also assist in reducing *viscosity*. Simple refactoring support, such as variable renaming, also reduces *viscosity*.

The DSVL uses a fairly *terse* set of simple color coded shapes with iconic annotations (e.g. a printer icon) as its basic notation. These are sufficient to cover the high level constructs needed, with a more *verbose* set of textual elaborations for lower level functions, for example, for reusable functions and meta-data links to the underlying Prism database. To mitigate the *diffuseness* and potential *error proneness* of these, features such as semantic checks that prevent elements and functions being used in inappropriate places are provided. This and the application of other syntactic and semantic constraints led to the almost complete elimination of spelling errors and other minor syntactic mistakes also reducing *error proneness*. Features, such as elision, support high level *visibility* and DSVL code *juxtaposition* at the expense of creating *hidden dependencies* to the elided code. This tradeoff was considered worthwhile to allow users to better obtain an overview of the report structure. Mechanisms such as co-selection of elements where other dependencies arose (e.g. specification/use of variables) served to reduce other types of *hidden dependencies*.

The two user evaluations targeted different end-user bases: 1) Prism RWL developers, i.e. those expected to maintain and extend the RWL meta-model (code base) and 2) End-users, i.e. those people regularly writing report specifications. For the developers, our focus was on the understandability and extensibility of the meta-model and associated visual notation, while for report authors our focus was on the usability of the prototype visual report specification

environment. Six developers and five end-users, all Prism employees, took part in these evaluations. The relatively small numbers meant qualitative information was sought. In both cases a brief tutorial was provided and the users then performed specified tasks and completed a survey.

Prism developers were asked to make a minor and a comparatively major modification to the meta-model using the newly developed meta-model. They were requested to record their steps and also respond to some close ended questions which were then analyzed to determine the expressiveness and usability of the meta-model.

The end-users were asked to design two reports using the newly developed visual language using the RWL modeling tool. They were also requested to record their steps and respond to some close ended questions which were then analyzed to determine the usability of the visual notation and the tool.

Developers were very positive about the meta-model used to capture the range of RWL concepts. They reported it made understanding and changing the meta-model much easier even for those who had little experience with the RWL semantics. Changing the meta-model and performing two extension tasks (part of our developer survey) also helped developers understand the meta-model and how the various constructs were related better. The meta-model helped end-users avoid making mistakes and also made it easier for the developer to define constraints as opposed to defining constraints in an arbitrary text based form. Developers found that linking visual elements and properties to the meta-model provides a consistent way to ensure semantically correct report models were created. Our model-driven approach [20] was seen as a good basis for generating other code if need be; such as configuration scripts.

However developers did perceive that meta-model scalability and navigation would be problematic as the model grew in size. There is a large learning curve involved with Microsoft DSL Tools and inexperience with model-driven development meant that the fundamental Model Driven Engineering (MDE) concept of generating code via transforming templates was not always intuitive to Prism developers.

Target report author end-users were particularly positive about the support the visual report writer language and tool provided for validation while designing Prism reports; they saw this as greatly saving time and effort. They found the Prism database browser for meta-data linking helped determine the current field or table to use accurately. The tool prevented report authors from making spelling mistakes and allowed them to obtain a clear overview of the database and its tables without understanding intimate details of its structure. Visually designing a

Prism report, based on existing Prism RWL concepts, allowed report authors to clearly visualize the report script and its logical flow. Participants noted that other report tools, whether focused on concrete or abstract end report layout, do not provide as good a control flow and element relationship visualization as our prototype. Large report changes are well-supported by the visual tool compared to textual RWL and other report tools.

However report authors found making trivial changes difficult using the visual notation. A small change such as adding a space in a line specification requires a number of steps as opposed to doing it textually. This confirms high viscosity at low levels of detail. There is also no synchronized view between the visual notation and the generated textual RWL script-concreteness. This results in poor *juxtaposibility*. Many end-users found the Visual Studio Shell (*Experimental Hive*) hosting the DSVL difficult to use. The visual notation was felt to be primarily suitable for novice and intermediate report designers as it does not allow a high degree of customization when compared to textually designing a report. Due to this reason expert report designers felt they would be constrained by the features of the tool and this may hamper their productivity.

Overall the Prism report developers found the model-driven approach to designing and building the visual report designer prototype very promising. Prism plans to further explore this approach for a deliverable visual report designer and to inform further development of the textual report scripting language and engine.

Novice and intermediate report designers found the visual reporting language and prototype support tool effective for increasing productivity and accuracy of report writing. Expert report users are not so well supported by the visual tool in its current form.

Future enhancements include adding incremental report generation and running, showing report designers the intermediate results of their evolving report design via *progressive evaluation*. A complementary WYSIWYG view showing report bands, literals and database content more closely matching the final report would not replace our existing visual notation but augment it. A visual debugger using the visual report authoring language to step through a running report would similarly aid complex report authoring and debugging.

7. Summary

We have developed a prototype visual report writing language and support tool using Microsoft Visual Studio DSL Tools for a commercial company in the print industry. This tool complements the existing

proprietary textual reporting language and engine for authoring custom enterprise system reports. Features of our approach include the use of model-driven development via meta-model, constraints, visual notation and code generators to realize the prototype visual language tool and enable much easier meta-model and visual language extension. Evaluations of both the model-driven engineering approach for visual languages and our prototype visual report designer language and tool suggest both are promising.

8. References

- [1] P. Panos and R. Weaver, "Factors Affecting the Acceptance of a Report Writer Software Application Within Two Social Service Agencies," *Journal of Technology in Human Services*, vol. 19(4), 2002.
- [2] D. Atkins, T. Ball, G. Burns, and K. Cox, "Mawl: a domain-specific language for form-based services," in *IEEE Transactions on Software Engineering*, vol. 25 (3), May/June 1999.
- [3] C. Pountney, *The Visual FoxPro Report Writer*. Hentzenwerke, 2002.
- [4] C. Weaver, "Coordinated Queries: A Domain Specific Language for Exploratory Development of Multiview Visualizations," in *IEEE Symposium on Visual Languages and Human-centric Computing*, Herrsching am Ammersee, Germany, September 2008.
- [5] R. Abraham, "FoXQ - XQuery by Forms," in *2003 IEEE Symposia on Human Centric Computing Languages and Environments*, Auckland, New Zealand, October 2003.
- [6] Taguchi, Mitsuhsa, and T. Tokuda, "A Visual Approach for Generating Server Page Type Web Applications Based on Template Method," in *IEEE Symposium on Visual/Multimedia Software Engineering*, Auckland, October 2003.
- [7] Microsoft. (2007) Microsoft. [Online]. [http://msdn.microsoft.com/en-us/library/bb126327\(vs.80.printer\).aspx](http://msdn.microsoft.com/en-us/library/bb126327(vs.80.printer).aspx)
- [8] Prism Group. (2008) Prism - Better information. Better business. [Online]. <http://www.prism-world.com/>
- [9] Prism Group, *Prism WIN - Report Writer Handbook*. Auckland, 2005.
- [10] M. Gunderloy. (2004, Jan.) A review of Crystal Reports V10 Advanced Developer Edition and MetaEdit+ 4.0, Application Development Tools. Internet.
- [11] Microsoft. (2008) Defining a Report Layout (Visual Studio Report Designer). World Wide Web.
- [12] Fast Reports Inc. (2009, Apr.) FastReport Studio. [Online]. <http://fast-report.com>
- [13] Perpetuum Software. Report Sharp Shooter. [Online]. <http://www.perpetuumsoft.com>
- [14] Papantonakis, Anthony, and P. J. H. King, "Syntax and Semantics of Gql, a Graphical Query Language," *Journal of Visual Languages and Computing*, vol. Special Issue on Visual Query Systems, Mar. 1995.
- [15] S. Di Martino, et al., "A WebML-based Visual Language for the Development of Web GIS Applications," in *IEEE Symposium on Visual Languages and Human-centric Computing*, Coeur d'Alene, Idaho, USA, 23-27 September 2007.
- [16] D. Dotan and R. Pinter, "HyperFlow: an Integrated Visual Query and Dataflow Language for End-User Information Analysis," in *VLHCC05*, Dallas, Texas, September 2005.
- [17] M. C. Humphrey, "A graphical notation for the design of information visualizations," *International Journal of Human Computer Studies*, vol. 50, no. 2, pp. 145-192, 1999.
- [18] A. Leff and J. Rayfield, "Relational Blocks: A Visual Dataflow Language for Relational Web-Applications," in *IEEE Symposium on Visual Languages and Human-centric Computing*, Coeur d'Alene, Idaho, USA, 23-27 September 2007.
- [19] M. Afonso, R. Vogel, and J. Teixeira, "From Code Centric to Model Centric Software Engineering: Practical case study of MDD infusion in a Systems Integration Company," in *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2006.
- [20] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*, . Springer, 2005.
- [21] S. Cook, G. Jones, S. Ken, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Boston: Addison-Wesley, 2007.
- [22] MetaCase. (2008) MetaEdit+ Modeler - Supports your modeling language. [Online]. <http://www.metacase.com/mep/>
- [23] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131-174, 1997.
- [24] J. C. Grundy, J. G. Hosking, K. Liu, and J. Huh, "Marama: an Eclipse meta-toolset for generating multi-view environments," in *ICSE08*, Leipzig, May 2008.

7.3 Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-driven Engineering

Almorsy, M. and Grundy, J.C. Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-driven Engineering, *1st ICSE Workshop on Software Engineering for High Performance Computing in Science (SE4HPCS 2015)*, Florence, Italy, May 19 2015, pp. 1-8.

DOI: [10.1109/SE4HPCS.2015.8](https://doi.org/10.1109/SE4HPCS.2015.8)

Abstract: Developing complex computational-intensive and data-intensive scientific applications requires effective utilization of the computational power of the available computing platforms including grids, clouds, clusters, multi-core and many-core processors, and graphical processing units (GPUs). However, scientists who need to leverage such platforms are usually not parallel or distributed programming experts. Thus, they face numerous challenges when implementing and porting their software-based experimental tools to such platforms. In this paper, we introduce a sequential-to-parallel engineering approach to help scientists in engineering their scientific applications. Our approach is based on capturing sequential program details, planned parallelization aspects, and program deployment details using a set of domain-specific visual languages (DSLs). Then, using code generation, we generate the corresponding parallel program using necessary parallel and distributed programming models (MPI, Open CL, or Open MP). We summarize three case studies (matrix multiplication, N-Body simulation, and digital signal processing) to evaluate our approach.

My contribution: Came up with initial ideas for the research, co-designed the solution, supervised the post-doc who implemented solution, lead investigator on grant funding the project from the ARC

Supporting Scientists in Re-engineering Sequential Programs to Parallel Using Model-driven Engineering

Mohamed Almorsy and John Grundy

Centre for Computing and Engineering Software and Systems
Swinburne University of Technology, Hawthorn, Australia
malmorsy@swin.edu.au, jgrundy@swin.edu.au

Abstract—Developing complex computational-intensive and data-intensive scientific applications requires effective utilization of the computational power of the available computing platforms including grids, clouds, clusters, multi-core and many-core processors, and graphical processing units (GPUs). However, scientists who need to leverage such platforms are usually not parallel or distributed programming experts. Thus, they face numerous challenges when implementing and porting their software-based experimental tools to such platforms. In this paper, we introduce a sequential-to-parallel engineering approach to help scientists in engineering their scientific applications. Our approach is based on capturing sequential program details, planned parallelization aspects, and program deployment details using a set of domain-specific visual languages (DSVLs). Then, using code generation, we generate the corresponding parallel program using necessary parallel and distributed programming models (MPI, OpenCL, or OpenMP). We summarize three case studies (matrix multiplication, N-Body simulation, and digital signal processing) to evaluate our approach.

Keywords—*component; Parallel Programming; High-Performance Computing; Domain-specific Visual Languages; Model-driven Engineering*

I. INTRODUCTION

The Australian Square Kilometer Array, which will enable astronomers to survey the radio universe with unprecedented speed, is expected to generate terabytes to petabytes of data per day of observations [1]. Processing such big data requires developing large-scale parallel programs that can fulfill the task and produce meaningful outcomes in a reasonable time. Before developing such parallel programs, scientists and HPC experts usually start with a sequential version that solves the problem on a small scale [2]. This is often relatively easy and helps to understand implementation details. However, scaling up such sequential programs to work on big datasets and utilizing the computational power of today's heterogeneous platforms is a very challenging task for scientists because it requires special experience in HPC. On the other hand, the existing parallel programming models and languages, such as MPI, OpenMP, OpenCL, are suitable mainly for expert parallel programmers. Existing automated and user-aided parallelization efforts try to address this gap. However, they are either very low-level, too abstract, or very domain-specific. We categorize these efforts in:

(i) *Compiler-based parallelization* [3-5]: Try to pinpoint parallelizable code sections, usually loop unrolling, in the input program either automatic using static analysis, machine learning and profiling techniques, or explicitly via user specified compiler directives. Kravets et al. [4] introduce Graphite-OpenCL that automatically locates parallelizable loops. Such loops are turned into an OpenCL kernel, and all necessary OpenCL calls for creating and compiling kernels and copying data to/from the device are automatically generated. Similar work was introduced by compilers including Polaris [5], and SUIF [3]. These efforts lack context and program developers' intention information.

(ii) *Abstract modeling or domain-specific languages*: Deliver high level models and/or DSLs usually have implicit mappings to predefined parallel libraries without letting developers specify intended parallelization details [6] such as image processing [7], partial differential equations (PDE) [8] or machine learning [9]. Although, these efforts help in hiding parallelization details from the user, they are limited by the provided functionalities by such DSLs. Moreover, most are text-based with a specific syntax that the developer has to learn to use the language.

(iii) *Portable domain-specific languages* [10-14]: Focus on capturing program parallelization aspects using models, then generating parallel code targeting one or more of the parallel programming models. Jacobo et al. [10] focus on using abstract models to help porting to different computing platforms – e.g. refactoring parallel programs to use MPI instead of OpenMP and switching between OpenCL and CUDA. Han et al. [11] introduce a directive-based approach where developers can annotate their sequential program code. This looks very similar the OpenACC APIs. Dig et al [12] introduce a refactoring tool that help in automating the conversion of sequential programs into parallel program without defining any annotations. The tool is based on locating signatures of possible parallelization aspects (mainly three aspects were covered including the recursion). The approach is based on replacing such matched program constructs with their corresponding java parallel library implementations. Palyart et al. [13] introduce MDE4HPC approach a DSVL to help in specifying and modeling HPC applications. They

focus on specification of solution parallelism. However, no code generation included. Jacob et al. also [14] introduce an IDE plugin to help programmers select code blocks and specify the computation device to run on. No support for heterogeneous computing platform.

In this paper, we introduce a new approach to help programmers and scientists in effectively parallelizing their sequential programs. Our approach is based on a set of domain-specific visual languages to help in modeling program structure and input/output datasets. Two further DSLs capture parallelization plans, and platform deployment details. Programmers/scientists do not need to have experience in parallel patterns and heterogeneous computing platforms. Using the parallel program model our toolset generates the necessary code utilizing these patterns and platform configurations. Generated code can be modified further, compiled and run on CPU grid and GPUs.

Our approach saves considerable effort required to migrate parallel programs from one computing platform to another. Thus, programmers and scientists can write a program, model its parallel and deployment aspects, and then get it to run on different computing platforms either single core, multi-core, many-core, or hardware accelerators by primarily updating program deployment details and having target code regenerated. We have evaluated our approach using several different scientific computation case studies including: general-purpose matrix multiplication, N-Body simulation, digital signal processing. Our approach is supported by a web-based tool that provides different features including: scientific DSL design, code generation, code editing, parallel patterns reuse, reverse engineering, and data visualization.

A. Motivating Example

Multiplication of large matrices is a common problem in scientific computing (e.g. 1 million elements each). Figure 1 shows a sequential program model of the matrix multiplication developed using our parallel program designer tool. It initializes both A and B (init_matrixA, init_matrixB) and then comprises three nested loops (Row, Cols, element – Loop) applying the multiplication operation on rows and columns to calculate the value of $C_{i,j}$. Finally, we print matrix_C. We discuss below how end users (scientists and programmers) using our approach can develop different parallel versions of such a program.

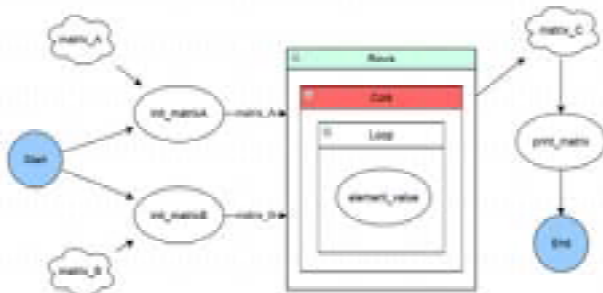


Figure 1. Example sequential matrix multiplication program

II. PARALLEL PROGRAM DESIGNER

Our approach is based on capturing a sequential program definition such as the example in Figure 1, then developing parallelization plans and deployment details. These details are used to generate the modeled parallel version of the program for further development. Our approach has three key DSLs, summarized in Figure 2: (i) SeqDSL: a Sequential program description language. This includes: sequence, selection, repetition, tasks and data structures. These constructs are available for users to model sequential algorithms as well as developing reusable tasks frequently used in a given domain. (ii) ParaDSL: Parallelization plan specification language. This includes operations necessary for data and task decomposition such as: split, join, parallel section, loop unrolling and parallel task. As discussed later, these basic constructs are sufficient in capturing parallelization plans used in different parallel patterns. (iii) DepDSL: Deployment details specification language. This includes computing platform, node specification, and groupings of nodes (in terms of communications). Currently, we capture basic information of the deployment platform and nodes such as number of nodes, grouping of nodes, how many cores per node, number of GPUs per node, etc. In this section, we discuss the parallel and deployment constructs. The sequential program specification is the same as in most of scientific workflows (further details are in [15]). Figure 1 shows an example sequential program model. Figure 2 shows the key concepts of our parallel program designer and key relationships. Below, we discuss these constructs, functionality and attributes.

A. Parallel Constructs

These constructs enable developers and scientists to specify how they plan to transform their sequential program into a parallel version. We focus on visually modeling possible task and data decompositions to achieve intended parallelization regardless of which parallel patterns or parallel programming models required to achieve such parallelization. Our constructs for parallelism are data decomposition (split and join), or task decomposition (parallel section, loop unrolling, parallel task).

- **Data Decomposition:** The first step in data decomposition is to think how input data could be split into smaller chunks where such smaller chunks can be processed faster and in parallel (divide-and-conquer). This implies that at some point we have to merge the outcomes of parallel tasks into one data structure.

Split construct divides an input data item into slices/chunks. The way this is implemented depends on the data structure being used. The user specifies whether a data item is geometrically divisible such as a 1D array, 2D array, image, cubes, hashtables, etc., or recursively divisible such as graphs or trees. For the first type, the user specifies what dimensions to use in splitting the data object – e.g. if we have a matrix, we can split in one dimension – e.g. rows or cols, or we could split in two dimension – i.e.

rows and cols (sub-matrices), and so on. Then, the user specifies the size of the slice: could be decided based on the number of processing nodes available (usually called BLOCK policy) or could be repeated slicing (CYCLIC policy) according to a selected slice size (every n elements form one slice). On the other hand, recursive data structures can be split using a link pointer (that points to a next list entry) and the number of computing entities – e.g. in a linked list, we may divide every consecutive N elements into a chunk or according to specific attribute value which is more expensive in terms of computations. The split operation usually does not exist separately. The outcome of the split operation depends on the next node – e.g. it depends on the parallelization technology used in next program node to decide how and where the new slices can be communicated to such nodes. The next node is a parallel section.

Join construct is the inverse of the split operation. It is usually preceded by a task parallelization construct (parallel section, loop unroll, or parallel task) that produces multiple slices of the intended outcome. The objective of the Join operation is to merge these pieces/results (generated by different threads or work items back to the output data structure. If the target data structure is a single-value variable, then the user should state a reduction operation to apply – e.g. if each node calculates the sum of an array chunk, then the Join will compute the sum of sums. In the other case (merge), users specify the target output variable.

- **Task Decomposition:** Program tasks can be parallelized either by decomposing a given task into multiple concurrent subtasks or applying the same task on small chunks of the input data, or both. The latter case usually involves breaking down (unrolling) task loops into subtasks where each subtask does less iterations.

Parallel Section construct helps in grouping a set of operations in one code block (visually, a container) that we want to consider as one unit for parallelization. Scientists can select, according to nature of the available resources, the parallel model to be used in realizing a parallel section – e.g. using multi-core (OpenMP), using GPUs (OpenCL), or distributed nodes (MPI). Data items passed to a parallel section should be broadcasted to the target processing

elements on which the parallel section runs. Copying data to/from a parallel section has different scenarios that we discuss below.

In a multi-core model: the parallel section is translated into a code block. The data items declared inside a parallel section or passed directly from outside nodes to an enclosed entity (there is an edge coming from a task to a task enclosed inside the parallel section) are considered as *private* to threads (a parameter in OpenMP directives), whereas parallel section passed in parameters (edges go directly to the parallel section) are considered as *shared* between all threads.

In clustered compute nodes: the parallel section is translated into a code block and the passed in data are distributed (if data are the outcome of a split operation), or broadcasted from the master node to slaves. The parallel section output data are copied from slave nodes back to the master node. This usually followed by a join operation to merge/reduce received data.

We can nest multiple parallel sections, each reflecting a parallelization level and different nested technologies may be used when realizing contained tasks. This is helpful when dealing with heterogeneous computing that requires both distributed (e.g. MPI) and shared memory models (e.g. OpenCL).

Loop Unrolling construct helps in realizing loop parallelism patterns that focuses on unrolling program loop iterations for execution by separate threads (in the case of multiple-cores), work items (in the case of GPUs), slave-based iterations (distributed nodes). Loop unrolling usually requires modifying loop header to run for fewer iterations.

Parallel Task construct is used in two cases: modeling tasks that are already parallelized and do not need to be revised by our code generator – e.g. readymade libraries, user defined tasks; and with tasks to be executed as they are in parallel - i.e. we just need to convert it into e.g. GPU kernel code and use it as it is. If the task needs to be revisited for parallelization, the user replaces it with a parallel section and flush task details and required parallelization.

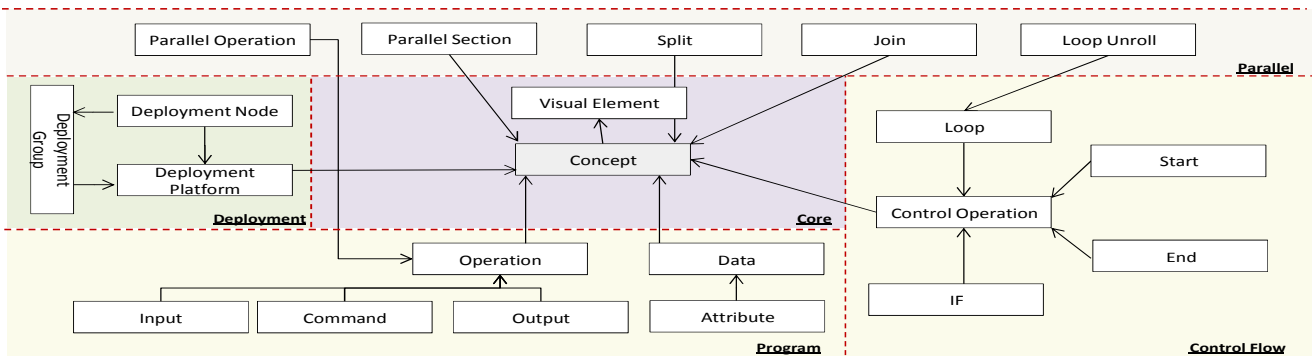


Figure 2. Parallel program designer constructs and their key groups and relationships

B. Deployment Constructs

These help developers and scientists to model a program's underlying compute platform. Currently, we are mainly interested in capturing how many nodes to be used in the deployment model, specifications of each node (memory, CPU, storage) and number of accelerators plugged in each node. Each node has a group name attribute that is used when grouping nodes into sub-clusters. This is helpful when mapping specific tasks for execution on a group of nodes – e.g. in the MapReduce model we have Map nodes and Reduce nodes. As a further extension of this we plan to include inter-node communication speed and bandwidth, and accelerator (e.g. GPUs) specification details. This helps in generating efficient code based on nodes capacity and tradeoffs between parallelization and communication overhead.

C. Model Refinement

A key problem with existing parallel programming models is the lack of a common model that can work with different, heterogeneous computing platforms. We have found many efforts proposing new common models that try to replace two or more existing parallel models with one platform [16]. In this section, we show that most of the parallel scenarios supported by these parallel programming models can be modeled using our approach.

Message-Passing Interface - (MPI): A set of APIs that facilitate communication between different nodes in a distributed memory based supercomputers or clusters. MPI has APIs for communication (send/receive), synchronization between different nodes (Barriers), combining results from different nodes (gather, reduce), and sharing public information about the cluster (number of processes, current process id). The Split (realized as a loop over number of nodes while calling send API to send data to all slave nodes and another piece of code in the slave node to receive the data slice to work on), Join (realized as send by slave node and loop in the master node to receive data), and program/data flow edges cover the communication (code to run on the master node and code in slave nodes (parallel section)) and reduction operations. The deployment group covers the possible communicators (communication groups). The parallel section helps in consolidating code blocks run in parallel on different nodes. Elements outside the parallel sections represent data and tasks to be executed on the master node. The deployment nodes define the number of nodes to run the MPI program on – i.e. in the `MpiRun` command params.

OpenMP: A set of compiler directives and APIs that help in parallelizing programs using multi-threading to utilize multi-core shared-memory architectures. OpenMP is based on the fork-join parallel pattern. Parallel Section can be used to group all instructions that we need to execute in parallel on multi-cores processors (using OpenMP). OpenMP usually has to specify the shared data between all threads and the private data for each thread. When multicore mode is selected: any data item declared inside a

Parallel Section is declared as a private memory, while data sent to the Parallel Section are considered shared data between threads. This is the same with Parallel Section outgoing edges. Finally, the reduction of data generated by all threads is done using the Join construct. Necessary platform information is delivered as parallel section attributes – e.g. number of threads, current thread id, etc. Our Loop Unrolling construct captures the OpenMP parallel loop directive.

Open Computing Language (OpenCL): A platform to help in heterogeneous computing using CPUs and/or GPUs. OpenCL is based on the Single Program Multiple Data (SPMD) computing model. It provides a set of APIs to help in creating kernels (programs) to be executed by GPU work items (threads). It also supports copying memory between host and accelerator device global memory. Another set of APIs provide current work item Id relative to local work group and global work items. To convert any program into the SPMD model, we have two options: if the tasks will be distributed, then we can leave the task unmodified and pass in a chunk of the data to each parallel thread or process. In the shared memory model we have to change the task to have each instance work on a portion of the *shared* data. Loop unrolling is usually a good candidate source of SPMD parallelism. In this case, we may consider each iteration as a separate thread (work item), so we replace the for loop header with a statement (loop variable = global thread id). Otherwise, we extend the loop header elements (initialization, condition, and update) with a dummy variable that counts how many iterations per thread – e.g. for (initial, `unroll var = 0`; condition `&& unroll var < number of iterations`; update, `unroll var++`). The Parallel section can be used to enclose elements that constitute a new kernel. All arrows directed into the parallel section are considered as kernel parameters and copied from host memory to device global memory. All arrows directed towards one of the internal operations in the parallel section are considered for further local memory copy. Parallel section also delivers global information such as `get_global_id` and `get_local_id`, etc. The user needs to specify the number of blocks and number of threads per block. These are parameters in the parallel construct.

Combined MPI, OpenMP, and OpenCL: When targeting heterogeneous platforms we can use nested parallel sections. The outer section could be used for e.g. distributed nodes (MPI), while the inner sections for multi-cores (OpenMP) or GPUs (OpenCL).

III. CASE STUDIES

To evaluate the effectiveness of our approach in parallelizing programs and improving developers' capabilities in handling parallel programs, we have conducted a set of case studies with several scientists. We summarize three of them here: matrix multiplication, N-Body simulation and digital signal processing.

A. Parallel Matrix Multiplication

In this section, we show how our approach can help in parallelizing the sequential matrix multiplication program from Figure 1. An initial thought to parallelize this program could be to split one of the matrices while keeping the other matrix as it is, or splitting both A and B. The later will have an impact on the calculations (sum of matrix A rows multiplied by matrix B cols). In either case, we need to take into consideration the deployment details: are we going to run this on a cluster or on a single node? Also we may have other computing devices – e.g. GPUs - that we might use in completing subtasks assigned to each node.

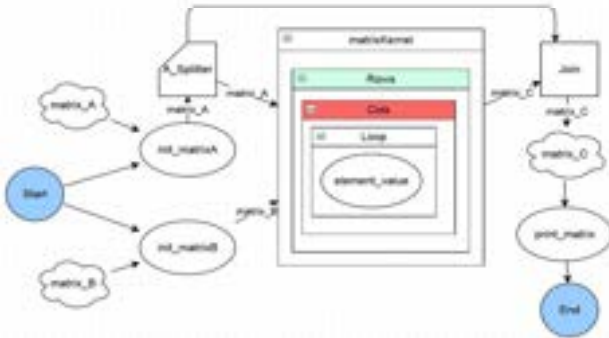


Figure 3. Example parallel matrix multiplication model

```

...
int taskid, ntasks;
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
...
//@Splitter: A_Split
int offset0 = 0;
int aSplitbulkSize = 1*aRows aCols/ ntasks ;
if( taskid == 0 ) {
for( int dest = 1 ; dest <= ntasks ; dest++) {
ierr = MPI_Send( A[offset0], aSplitbulkSize , MPI_DOUBLE, dest
, 0 , MPI_COMM_WORLD);
offset0 = offset0 + bulkSize;
...
else if( taskid != 0 ){
ierr = MPI_Recv( A, aSplitbulkSize, MPI_DOUBLE, 0 ,
0 , MPI_COMM_WORLD , &status); ...
//@Parallel Section: MatrixKernel
if( taskid != 0 ) { // Calculate C[i][j] }
//@Join: Join
if(taskid !=0) {
ierr = MPI_Send(&C[offset0], aSplitbulkSize, MPI_DOUBLE, 0 , 0 ,
MPI_COMM_WORLD); ... }
else if( taskid == 0 ) {
for( int i=1; i<=ntasks; i++) {
ierr = MPI_Recv(C[(i -1) * aSplitbulkSize], i * aSplitbulkSize, MPI_DOUBLE, i ,
0 , MPI_COMM_WORLD , &status); } ...

```

Figure 4. A snippet of generated parallel matrix multiplication code modeled in Figure 3

Figure 3 shows a scenario where we decided to split matrix A and distribute the slices to different cluster nodes and broadcast matrix B to all nodes. This should be done on the master node (tasks outside the parallel section “matrixkernel”). The parallel section defines that the enclosed tasks (loops & calculation) are to be executed in parallel on slave/worker nodes. The outcome of the matrix multiplication by each node is then sent back to the master node where we have a Join operation to merge results

together into matrix_C. We have configured the *splitter* construct properties to split matrix A using *block_policy* according to number of nodes and then distribute each chunk to slave nodes. Another splitter could be added if we want to split matrix B as well, but will impact the calculations operation. Figure 4 shows a snippet of the generated parallel code for matrix multiplication. We show parts of the generated code reflecting the split, join and parallel section realization. The rest should be the same as in the sequential version. Using split and join with a cluster computing model is realized using the MPI master-slave pattern (one of the well-known parallel design patterns).

```

//@Parallel Section: GPUKernel
cl_program program;
cl_kernel kernel;
const char *KernelSource = "\
__kernel void ParallelSection( __global double* A, __global double* B) {\
int i = get_global_id(0); {\
for(j = 0; j < aCols; j++) {\
calculate_matrix_element();\
}}";
program = clCreateProgramWithSource(context, 1, (const char **) &
KernelSource, NULL, &err);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "ParallelSection", &err);
cl_mem memobjA = clCreateBuffer(context, CL_MEM_READ_WRITE, *, NULL,
&ierr);
ierr = clEnqueueWriteBuffer(command_queue, memobjA, CL_TRUE, 0, *, A, 0,
NULL, NULL); ...

```

Figure 5. A code snippet generated for GPU kernel

Now, assume that each computing node has a GPU device so that we can use it to accelerate the computations assigned to nodes. In this case we need to add another parallel section inside “matrixkernel” and select “Run on GPU” property to encapsulate the computational tasks into *kernel* to be computed by GPU work groups and work items. The number of work items, work groups, and the dimensions of work items in work groups depend on the size and dimensions of the data being processed. Moreover, the data to be processed by the GPU device will be copied from the host node memory to GPU global memory. The same will be done after work items finish.

Table 1. Performance results (in sec) of the generated MPI code with different matrix size and number of nodes

Matrix Size	1 node	3 nodes	5 nodes	7 nodes
100 X 100	0.04	0.01	0.0017	0.0014
300 X 300	0.16	0.04	0.025	0.017
500 X 500	1	0.19	0.11	0.06

Table 2. Performance results (in sec) of the generated MPI+OpenMP code with different matrix size, nodes, cores

Matrix Size	1 node	3 nodes	5 nodes	7 nodes
100 X 100	0.04	0.006	0.008	0.005
300 X 300	0.16	0.03	0.016	0.01
500 X 500	1	0.14	0.09	0.04

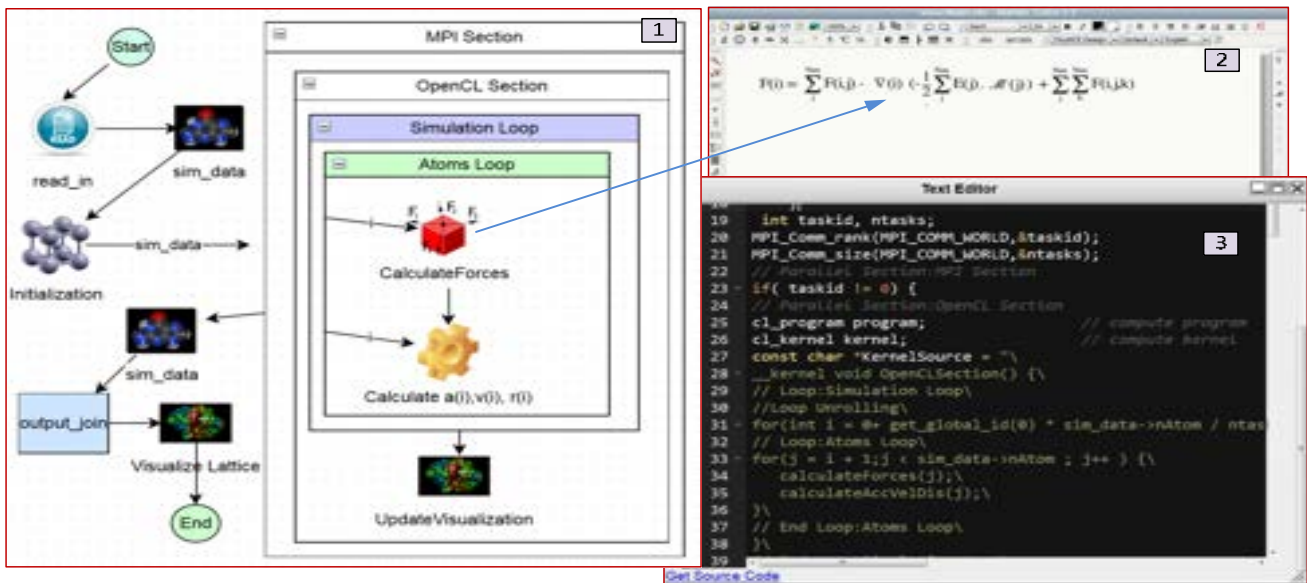


Figure 6. N-Body simulation models in our web-based IDE tool, Horus HPC

Tables 1, 2 shows the performance improvement (in secs) of the generated code using MPI and MPI+OpenMP against the sequential version of the matrix multiplication program. The time reported excludes matrices initialization tasks, because it is common in all scenarios. It is worth mentioning that the performance of the generated code depends heavily on the user-modelled transformation. We plan to build new plugins that can generate recommendations regarding the parallelization plan and number of nodes, blocks and threads to use.

B. N-Body Case Study

In this case study we asked a new PhD student who is working on molecular simulation to use our tool to transform her sequential program into MPI & OpenCL version. She did not have any previous experience in parallel programming. We conducted a 1-hour training to explain how the approach works. *First*, we asked her to model her sequential program using our sequential DSVL developed earlier (introduced in [15]). Then, we asked her to use the parallel construct to model how she wanted to parallelize her program. Figure 6-1 shows a snapshot of the parallel model. Figure 6-3 shows a part of the parallel program, using MPI and OpenCL, generated by our tool based on parallel model Figure 6-1. This took around four hours. On the other hand, we asked a HPC programmer to parallelize the same N-Body sequential program using, which was written by someone else, using MPI and OpenCL. The programmer reported that the initial working parallel version that is the same as our parallel model took around three weeks full-time. It took the same time to get an optimised version.

C. DSPSR Case Study

The third case study is reengineering the detection module in an existing digital signal processing for pulsar astronomy tool developed internally at Swinburne by

astrophysics team (<http://dsp.sr.sourceforge.net/>). The core code of the detection module, a part of it is shown in Figure 7, has mainly two nested loops: one on input signal channels (nchan=512) and the second nested loop is on input signal periods (ndat=1024) for every pairs of signals. The intended solution was to unroll both loops on 8 nodes (each work on 512/8=64 iterations). Each node has a GPU (should work on the inner loop, 1024 iterations). The solution was using two nested parallel section constructs: one for node distribution (MPI), and the other for GPU processing. The key problem we faced in this parallelization task was the dependency between iterations to advance array pointers. The scientist made a simple modification to link array indexes to loop variables instead of using pointers. Then we unrolled the inner loop in an OpenCL GPU kernel.

```

for (unsigned ichan=0; ichan<nchan; ichan++) {
    const float* p = input_base + ndat*ndim*npol*ichan;
    const float* q = p + ndat*ndim;
    r[0] = output_base + ndat*ndim*npol*ichan;
    r[1] = r[0] + 1;...
    for (j=0; j<ndat; j++) {
        p_r = *p; p++;
        p_i = *p; p++;
        pp = p_r * p_r + p_i * p_i;
        qq = q_r * q_r + q_i * q_i;

        *S0 = pp + qq; S0 += span;
        *S1 = pp - qq; S1 += span;
        ...
    }
}

```

Figure 7. excerpt from the detection module to be reengineered

IV. IMPLEMENTATION

Figure 8 shows a high-level architecture of our toolset, Horus HPC, with its two main components: the model designers and code generators. Our parallel program designer is implemented using our Horus framework as a web application based on HTML5 and JavaScript [15]. It is

built on an existing open source web modeling tool (<https://www.draw.io/>). The web-based platform allows developers and scientists to use, collaborate, and share models and solutions. We extended this tool with a DSL designer to help in developing domain-specific visual languages (DSLs). This helps scientists and developers in creating operations and data structures that are common to their domains without a need to redefine such operations or data structure when modeling new problems. Once a DSL is defined, users can register it to be used from within the tool itself.

The parallel program designer (Figure 8-1) is implemented as a DSL with all constructs reflected in the meta-model discussed above. The code generator (Figure 8-2) is implemented in JavaScript to complement the parallel DSLs provided through our web-modeling tool. It simply parses the graph nodes (vertices and edges). For simple nodes in the program (those with no parallel impact, e.g. data, if, or task nodes), the core code generator outputs the corresponding realization code, which is currently in C language. For parallel operations – e.g. split, join, and parallel sections, etc. the code generator consults the corresponding parallel technology code generator – e.g. in a split node, if the distributed flag is set, then the MPI code generator is triggered, a sample MPI code generator is shown in Figure 9. This code snippet shows parts of the MPI initialization code, and split construct code generator. Otherwise, if the shared flag is set in a parallel section, the OpenCL generator is triggered. The resultant parallel program is then compiled and deployed on the target computing platform (Figure 8-3). To support reverse (and round-trip) engineering we use code annotations (The same annotation used in the generated code in Figure 4 and Figure 5): where scientists can add annotations in code to guide the model reconstruction process.

Our code generation component takes program models, as a set of nodes and edges developed with the parallel program designer discussed above, as input and generates a corresponding program with the necessary parallelization aspects as defined by users. The code generator has a sequential code generator as its core. This core code generator is responsible for generating a complete sequential version of the user specified model (without any parallelization). This is helpful when the user would like to build his initial program from the existing (predefined, or other users' defined) building blocks. For parallelism, we have separate generators for different parallel programming model – i.e. we have MPI, OpenMP, OpenACC, and OpenCL code generators. This permits further improvements of such code generators separately without modifying the whole code generation component.

The core code generator will issue calls to those specialized code generators whenever it finds a parallel construct used in the input model. According to the parallel model used in these constructs, the core code generator calls the corresponding generator – e.g. if a parallel

construct is used with distributed model, the MPI code generator is called. The outcome of this code generation is weaved within the sequential program as needed. This approach improves extensibility in order to support further possible programming models and extension of individual code generators. Each code generator processes each construct in the designer meta-model taking into consideration that the meta-model may be extended in future (more details in the implementation section).

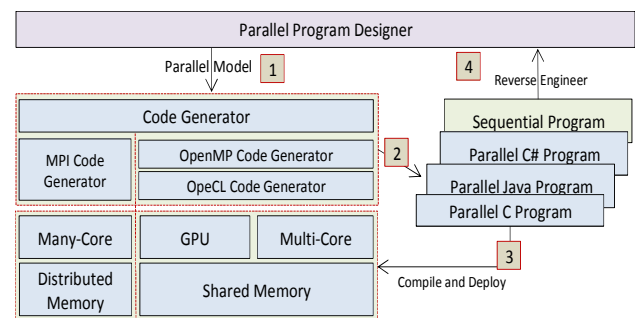


Figure 8. A high-level architecture of our approach

```

...
MPICodeGenerator.prototype.Start = function(currentNode, graph, output, flags) {
...
output.append(" int taskId, ntasks;");
output.append("MPI_Comm_rank(MPI_COMM_WORLD,&taskId);");
output.append("MPI_Comm_size(MPI_COMM_WORLD,&ntasks);");
}
MPICodeGenerator.prototype.Split = function(currentNode, graph, output, flags) {
...
output.append(" int bulkSize = " + copySize + ";");
output.append("if( taskId==0) {\n");
output.append("  tfor (int dest = 1 ; dest <= ntasks ; dest++) { \n");
var line = 'ierr = MPI_Send(' + propertyName + copyIndex + ', bulkSize , '
          + dataType + ', dest , 0 , MPI_COMM_WORLD);';
...
}

```

Figure 9. Code snippet of the MPI Code Generator

First, the code generator checks the specified deployment details: Is it a single node? a cluster? Do nodes have accelerators(GPUs)? Do we have multi-core CPUs? The answers to these questions are used in initializing the program runtime environment including MPI and/or OpenCL initialization as described in the fourth step. *Second*, we generate declarations for all data nodes to avoid “undeclared identifier” compilation errors that may arise when generating tasks that depend on data not declared in the program yet. This usually happens when a node has multiple edges and the edge order does not reflect the real dependency of the next nodes. *Third*, the core code generator builds a dependency list of the model nodes. This list is used to decide the order of graph nodes and edges to traverse for code generation. The code generation starts from the graph *start* node and keeps generating code for the rest of the graph elements until all nodes have been *realized*. *Fourth*, the code generator issues a call to technology-specific code generator according to the node currently being processed and technology reflected on the node, e.g. if the node is a start node, necessary code generators are called to perform necessary initialization tasks. If the node is a split with distributed memory model,

then the MPI code generator is called. The same applies on Join, Parallel section, and Loop graph nodes. Each function in the code generator has a predefined signature as follows (currentNode, programGraph, outputCode, flags). Figure 9 shows a code snippet from MPI code generator.

V. THREATS TO VALIDITY

The number of experiments we have conducted so far is limited with regard to usability, expressiveness, performance and extensibility aspects. We plan to conduct more detailed performance and usability evaluations.

Our code generation approach has some limitations, which we are currently working to address: (i) Memory Management: currently, we do not address optimized memory access when generating kernel code (memory coalescing techniques). (ii) Support for custom data type data communicated between nodes when using MPI. However, this should be easy to support. The same with OpenCL data types such as float2, float4, etc. Currently, this must be specified by developers. (iii) Recursive functions. This is a limitation of the underlying technology – e.g. CUDA does not support recursive kernels. (iv) Data dependency between parallelized iterations is not automatically resolved. It is still the developer/scientist responsibility to handle data dependencies. (v) We do not make use of the parallel programming platform special libraries such as CUDA cuFFT and cuBLAS-XT. This is a design limitation of our current approach implementation. We do not modify the core code of the sequential program. We do extend such code with parallel constructs. Users can add *parallel task* model element that refers to CUDA libraries. We plan to extend our parallel code generators with a collection of libraries and when to use them. Thus, the code generator can look for matches to replace with parallel library APIs. (vi) Deep modifications of the enclosed tasks to use `threadId` or `work item Id` is not supported. The readability of the generated code could facilitate fine-tuning activities.

VI. SUMMARY

We described a novel model-driven approach to help in transforming sequential programs to parallel versions through visualizing parallelization aspects and patterns. Our approach enables utilizing the underlying computing platforms without deep experience in parallel programming models. We capture sequential program details including data structures and tasks. Scientists and developers then extend this model with parallelization specifications and specify the deployment details of the program. These three aspects are used in generating a parallel version of the input sequential program. This approach facilitates updating any or all of these three aspects without modifying the other aspects in the model. Code generation supports different possible deployment models and programming models including distributed nodes using MPI, multi-cores using OpenMP, and GPU accelerator devices using the OpenCL and OpenACC programming model. We have applied our

approach to several problem domains. Due to space limitations only three examples discussed in this paper including matrix multiplication, N-Body simulation and Digital Signal Processing.

VII. ACKNOWLEDGMENT

This research is supported by the Australian Research Council under Discovery Project DP120102653.

REFERENCES

- [1] S. Johnston, M. Bailes, N. Bartel, C. Baugh, M. Bietenholz, C. Blake, *et al.*, "Science with the Australian square kilometre array pathfinder," *Publications of the Astronomical Society of Australia*, vol. 24, pp. 174-188, 2007.
- [2] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming*: Pearson Education, 2004.
- [3] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, *et al.*, "Maximizing Multiprocessor Performance with the SUIF Compiler," *Computer*, vol. 29, pp. 84-89, 1996.
- [4] A. Kravets, A. Monakov, and A. Belevantsev, "GRAPHITE-OpenCL: Generate OpenCL Code from Parallel Loops," presented at the GCC Summit 2010, 2010.
- [5] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, *et al.*, "Automatic Detection of Parallelism: A grand challenge for high performance computing," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 2, p. 37, 1994.
- [6] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, *et al.*, "Composition and Reuse with Compiled Domain-Specific Languages," in Proc. European Conference on Object-Oriented Programming, Montpellier, France, 2013.
- [7] J. Ragan-Kelley, C. Barnes, *et al.*, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in Proc. 34th ACM Conf. on Programming language design and implementation, Seattle, USA, 2013.
- [8] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, *et al.*, "Liszt: a domain specific language for building portable mesh-based PDE solvers," presented at the Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, Washington, 2011.
- [9] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, *et al.*, "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning," presented at the Proceedings of the 28th International Conference on Machine Learning, 2011.
- [10] F. Jacob, R. Arora, P. Bangalore, M. Mernik, and J. Gray, "Raising the level of abstraction of GPU-programming," in Proc. 16th Int. Conf. on Parallel and Distributed Processing, pp. 339-345, 2010.
- [11] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, D.C., 2009.
- [12] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in Proceedings of the 31st International Conference on Software Engineering, 2009.
- [13] M. Palyart, D. Lugato, *et al.*, "HPCML: A Modeling Language Dedicated to High-Performance Scientific Computing," in Proc. of 1st Int. Workshop on Model-Driven Engineering for High Performance and Cloud computing, Innsbruck, Austria 2012.
- [14] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "CUDAACL: A tool for CUDA and OpenCL programmers," in Proceedings of 2010 International Conference on High Performance Computing (HiPC), 2010, pp. 1-11.
- [15] M. Almorisy, J. Grundy, *et al.*, "A Suite of Domain-Specific Visual Languages For Scientific Software Application Modelling," in the proceedings of 2013 IEEE Symposium on Visual Languages and Human-Centric Computing, San Jose, CA, USA, 2013.
- [16] J. Diaz, C. Munoz-Caro, and A. Nino, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1369-1386, 2012.

7.4 A visual language and environment for enterprise system modelling and automation

Li, L, Grundy, J.C., Hosking, J.G. A visual language and environment for enterprise system modelling and automation, *Journal of Visual Languages and Computing*, vol. 25, no. 4, April 2014, Elsevier, pp. 253-277
DOI: [10.1016/j.jvlc.2014.03.004](https://doi.org/10.1016/j.jvlc.2014.03.004)

Abstract: Objective: We want to support enterprise service modelling and generation using a more end user-friendly metaphor than current approaches, which fail to scale to large organisations with key issues of “cobweb” and “labyrinth” problems and large numbers of hidden dependencies. Method: We present and evaluate an integrated visual approach for business process modelling using a novel tree-based overlay structure that effectively mitigate complexity problems. A tree-overlay based visual notation (EML) and its integrated support environment (MaramaEML) supplement and integrate with existing solutions. Complex business architectures are represented as service trees and business processes are modelled as process overlay sequences on the service trees. Results: MaramaEML integrates EML and BPMN to provide complementary, high-level business service modelling and supports automatic BPEL code generation from the graphical representations to realise web services implementing the specified processes. It facilitates generated service validation using an integrated LTSA checker and provides a distortion-based fisheye and zooming function to enhance complex diagram navigation. Evaluations of EML show its effectiveness. Conclusions: We have successfully developed and evaluated a novel tree-based metaphor for business process modelling and enterprise service generation. Practice implications: a more user-friendly modelling approach and support tool for business end users.

My contribution: Co-developed key ideas for this research, co-designed approach, co-supervised PhD student, wrote substantial parts of paper, co-lead investigator for funding for this project from FRST

A visual language and environment for enterprise system modelling and automation

Lei Li
Beef & Lamb New Zealand
PO Box 121
Wellington 6140, New Zealand
Richard.Li@beeflambnz.com
Ph: +64-4-471-6035

John Grundy
Centre for Computing &
Engineering Software Systems
Swinburne University of
Technology
PO Box 218, Melbourne, Australia
jgrundy@swin.edu.au
Ph: +61-3-9214-8731

John Hosking
College of Engineering and
Computer Science
Australian National
University
Canberra, Australia
john.hosking@anu.edu.au
Ph: + 61-2-6125 8807

Abstract

Objective: We want to support enterprise service modelling and generation using a more end user-friendly metaphor than current approaches, which fail to scale to large organizations with key issues of "cobweb" and "labyrinth" problems and large numbers of hidden dependencies. **Method:** we present and evaluate an integrated visual approach for business process modelling using a novel tree-based overlay structure that effectively mitigate complexity problems. A tree-overlay based visual notation (EML) and its integrated support environment (MaramaEML) supplement and integrate with existing solutions. Complex business architectures are represented as service trees and business processes are modelled as process overlay sequences on the service trees. **Results:** MaramaEML integrates EML and BPMN to provide complementary, high-level business service modelling and supports automatic BPEL code generation from the graphical representations to realise web services implementing the specified processes. It facilitates generated service validation using an integrated LTSA checker and provides a distortion-based fisheye and zooming function to enhance complex diagram navigation. **Evaluations of EML show its effectiveness.** **Conclusions:** we have successfully developed and evaluated a novel tree-based metaphor for business process modelling and enterprise service generation. **Practice implications:** a more user-friendly modelling approach and support tool for business end users.

Keywords: business process modelling, web service generation, process enactment, zoomable user interfaces, domain-specific visual languages, business process modelling notation, business process execution language

1. Introduction

Business processes play a crucial role in running an organisation. In order to achieve process excellence, people carry out various process improvement initiatives, such as business process reengineering (BPR) and business process management (BPM) (Hill and Brinck et al 1994; Aguilar-Savén 2004), along with business process-based services (Li et al, 2010) and service composition (Gillain et al, 2013). BPM has been defined as "a structured, coherent and consistent way of understanding, documenting, modelling, analyzing, simulating, executing and continuously changing end-to-end business processes and all involved resources in light of their contribution to business success" (Recker 2010). BPM covers the overall management of organizations by looking at the lifecycle of their business processes (Box and Cabrera et al 2006; Kramer and Herrmann 2007). However, no matter which process improvement initiative people wish to conduct, they need to understand business processes, perform analyses to design or redesign them, and build or reuse appropriate services or service orchestrations to realise them (Jung and Cho 2005; Grundy et al 2006).

BPM and business process-based service composition have thus emerged as popular approaches in practice and research. However, while organizations appear to be well aware of the need for BPM and the advantages of using service-oriented architectures, implementation remains a challenging task. Indeed, many organizations still struggle with an efficient modelling approach to discover, visualize and document their business processes (Recker and Rosemann 2009). Examples of BPM approaches include Entity-Relationship models (Chen. 2002), Data Flow Diagrams, Flow Charts (Urbas and Nekarsova et al 2005), Scenarios, Use Cases, Integration Definition for Functional and workflow Modelling (Eriksson and Penker 2000), and Business Process Modelling Notation (BPMN) (BPMI 2010). Many types of workflow management and service modelling and

composition systems have been developed to model and implement enterprise business processes (Recker et al 2009; Paussto 2005; Leymann 2001; Xiong et al, 2010). Their goal is to specify, compose, co-ordinate, enact and evolve business processes using a high-level visual modelling approach. Using workflow approaches, business processes are typically modelled as stages, tasks and links. These models are then used to control the execution of software components that comprise an enterprise system. Process technology can also be used to model processes executed within systems e.g. in Enterprise Resource Planning (ERP) systems.

Despite this ongoing proliferation of process modelling languages, only a few have been widely accepted by practitioner communities. Research shows that visual process modelling methods differ significantly in their features and characteristics, such as, for instance, their representational capabilities, their support for expressing workflow patterns or their support for formal analysis of correctness criteria (Engels and Erwig 2005; Gamma et al 1995). Actual practice, on the other hand, informs us that certain BPM languages have achieved higher levels of adoption and dissemination in visual modelling practice than others while many visual modelling languages exist as objects of interest only to academic scholars (Gottfried and Burnett 1997; Grundy et al 2006). Most existing workflow based Enterprise visual modelling languages adopt box-and-line style diagrams, which work well for small to medium diagrams. A common difficulty with such approaches is a lack of scalability. Most existing modelling technologies are effective in only limited problem domains or have major weaknesses when applied to large system models resulting in “cobweb” and “labyrinth” problems, where users have to deal with many cross diagram flows. Most modelling tools use multi-view and multi-level approaches to mitigate this problem (Grundy et al 2000). These approaches have achieved some success but do not fully solve the problem, as using the same notation and flow method in a multi-view environment just reduces individual diagram complexity, but increases hidden dependencies between diagrams. This requires use of the long-term memory of the users, as they have to build and retain the mappings between views mentally. In addition, most existing flow based business modelling notations lack multiple levels of abstraction support.

We have been developing a new approach, Enterprise Modelling Language (EML) and a support environment, MaramaEML, to overcome some of these issues in a novel way (Li et al 2007; Li et al 2008). Our principal goal that directed the design of EML was to provide a simple, intuitive and executable visual notation to support rapid, user-friendly development of business process-based services. Our key target end-users are business process domain experts and service composition experts. EML adopts several visual metaphors to enhance the representation, navigation and management of large organizational hierarchies and process flows. It attempts to address many of the above limitations by modelling processes primarily by a novel tree overlay structure. In this new approach, complex business systems are represented as service trees and business processes are modelled as process overlay sequences on the service trees. By combining these two mechanisms, EML gives users a clear overview of a whole enterprise system with business processes modelled by overlays on the same view. However, our approach does not exclude existing modelling notations such as BPMN. We incorporate them in our EML support tool while providing additional richer, integrative views for enterprise process modelling. The objective of EML is to support business process management by both technical users and business users by providing a novel tree overlay based notation that is intuitive to business users yet able to represent complex process semantics. MaramaEML is an Eclipse-based integrated design environment for creating EML specifications. This IDE provides a platform for efficient visual EML model creation, inspection, editing, storage, model driven code generation of web services, and integration with other diagram types. Distortion-based fisheye and zooming functions have also been implemented to enhance MaramaEML’s navigability for complex diagrams. MaramaEML supports BPEL code that is automatically generated from graphical EML representations and mapped to a Labelled Transition System Analyser (LTSA) (Foster and Magee et al 2003) for validation.

The remainder of this paper is organized as follows: Section 2 describes the motivation for our research and Section 3 provides an overview of the approach taken and its development. Section 4 introduces the detailed design of the Enterprise Modelling Language (EML), describing the visual representations of service tree structure, process overlay and exception handlers; as well as some more advanced constructs such as dependency / trigger, iteration and conditions. This section uses a complex business process example (a University enrolment system) to demonstrate the capabilities of EML and MaramaEML. The architecture and implementation of MaramaEML are discussed in section 5. Section 6 described a formal user evaluation of EML and MaramaEML with analysis of the evaluation feedback and possible improvements. We conclude the paper with a discussion and future work directions.

2. Motivation and Related Work

Our primary motivation for this research came from an attempt to model a large university enrolment system as part of a process improvement exercise, and to derive executable service orchestrations from these models to be enacted by a workflow system. This is a complex service-oriented enterprise system that involves dynamic collaborations among five distinguished parties: Student, Enrolment Office, Department, Finance

Office and StudyLink (the New Zealand government’s student loan agency). The main functional requirements of the system are:

1. Students will use this system to search the course database and apply for enrolment in target courses; if their application is approved, they need to apply for a loan from StudyLink;
2. After receiving student applications, the Enrolment Office checks academic conditions with academic Department staff and then informs Students of the results;
3. Department staff check course enrolment conditions and make the final decision (approve or reject);
4. For an approved enrolment application, the Finance Office tracks fee payment and informs the Enrolment Office and Department of any changes. If a Student applies for a loan, the Finance Office also needs to confirm the student information with StudyLink.
5. StudyLink investigates the student information supplied by the university and then approves (or declines) the loan application.

A Business Process Modelling Notation (BPMN) diagram capturing some of this enrolment process is shown in Figure 1. This illustrates the use of process stages, “pools”, process flow, etc. when modelling a process. Unfortunately as the process definition grows, the user must create either massively complex and unwieldy diagrams or “drill down” into sub-stages, introducing hidden dependencies and complex navigation (Baker 2002; Recker and Rosemann et al 2009; Adams 2011). While BPMN, like many approaches, provides mechanisms to manage complexity via sub-processes, these introduce potentially severe hidden dependency and lack of juxtaposition problems (Genon et al, 2011).

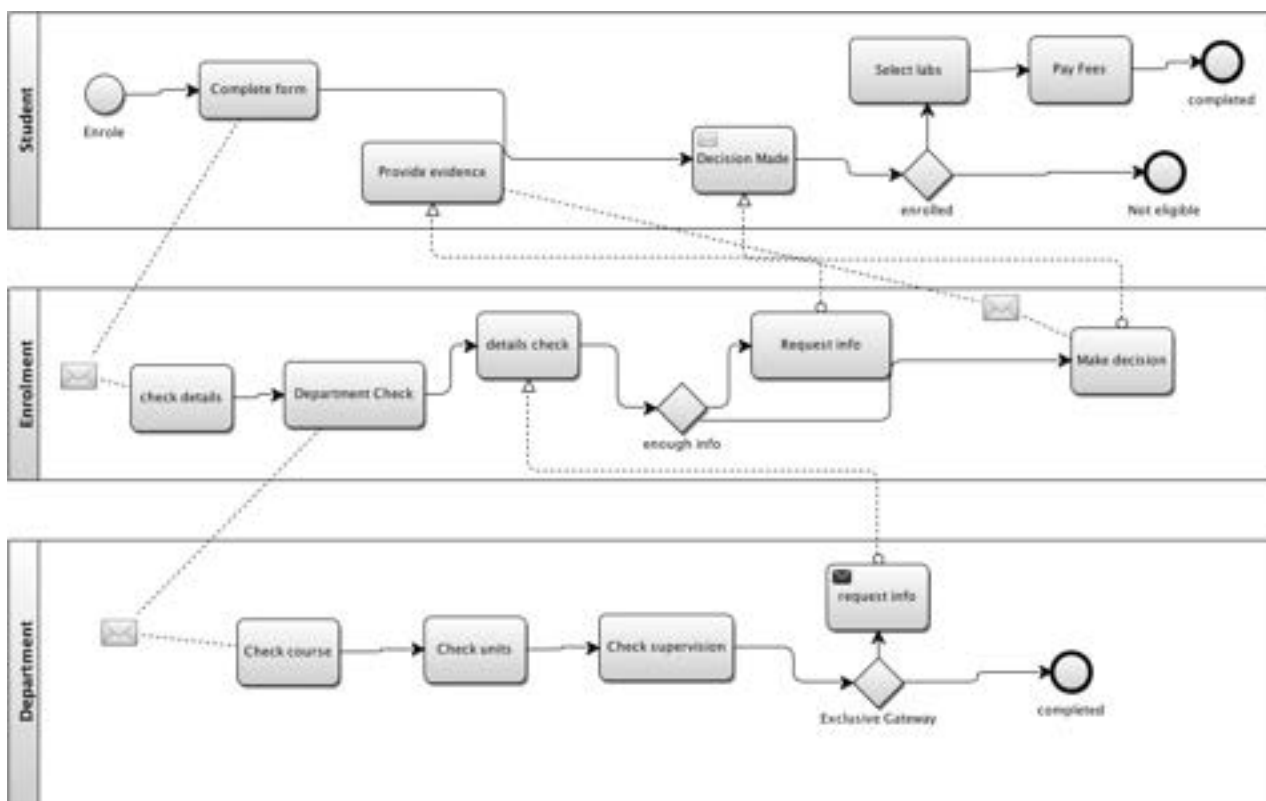


Figure 1: Part of a BPMN specification of the Enrolment System.

General-purpose modelling languages like UML (Schnieders and Puhlmann 2005) and Petri Nets (Marshall 2004) have a well-established set of modelling notations and constructs. Though they are sufficiently expressive to model business scenarios, they are difficult for a business user to learn and use. Domain specific languages like Web Transition Diagrams (WTD) and T-Web systems (Kornkamol and Tetsuya et al 2003) are very easy to understand but are limited to the scope of service level composition and modelling. They are not efficient in presenting multi-level abstractions of business processes. Business oriented frameworks like ARIS (Goel 2006) and TOVE (Buschmann and Rohnert 1996) are based on a generic and reusable enterprise data model technology and also provide a holistic view of process design. However they focus too much on technical processes and efficient software implementations and hence they can result in ambiguity of the models as extra programming knowledge is required.

Some efficient modelling languages like BPMN (BPMI 2010), BioOpera (Pautasso and Alonso 2005), YAWL (Adams et al 2011), Form Charts (Draheim and Weber 2005) and ZenFlow (Martinez and Patino 2005)

use visual notations to represent processes. Many also provide support tools to automatically generate industry standard code like BPML and BPEL4WS (BPMI 2006) meaning the notations are, in some sense, executable. They all use workflow-based box and line methods to describe the system. Severe cobweb and labyrinth problems appear quickly using this type of notation to model the enrolment system (Recker and Rosemann et al 2007). Nordbotten and Crosby believe that graphical complexity plays a key role in the usability of any visual language (Nordbotten and Crosby 1999). The main principle of Graphical Complexity states that the complexity of a visual notation should be cognitively manageable. Novices are much more affected by graphic complexity than experts, as they must consciously maintain meanings of symbols in memory. The more symbols there are to remember, the greater the cognitive load and potential errors. Many of these languages, including BPMN, have a very large number of symbols.

A related feature of modelling languages is the use of colour. Winn (Winn 1993) showed colour is one of the most cognitively effective of all visual variables in visual languages. Differences in colour are detected three times faster than shape and are more easily remembered. However, we also understand that colour is one of the most difficult variables to use effectively. If not used carefully, it can undermine communication (Moody 2009). BPMN does not prohibit the use of colour, but just not actively encourage the user to use colours. In BPMN every user can freely choose their preferred colour, so normally they vary from user to user, potentially leading to misinterpretations.

Another interesting characteristic of modelling languages is how readily they permit users to hand draw a model without using a software tool support. In the business process modelling area, this can be important as process models are typically developed in an interactive manner by sketching on whiteboards or paper. In fact, the original BPMN charter (BPMI 2010) stated that “If small BPMN processes cannot be easily jotted down by a business analyst on a blank piece of paper, then BPMN will not be successful.”

Diagram complexity management is often measured by the ability of notations to present large amounts of information without overloading the human mind. Citrin has described diagram complexity management as one of the most intractable issues in visual notations: “a well-known problem is that they do not scale well” (Citrin 1996). The limitation of diagram related elements that can be effectively processed by the human mind is limited by working memory capacity. When the limitation is exceeded, the cognitive overload issue appears rapidly. Moody has pointed out that BPMN lacks effective mechanisms for managing diagrammatic complexity (Moody 2006). The issue of complexity management is not addressed in the BPMN specification, suggesting that it was not considered in the design of the notation. Almost all the examples in the BPMN specification and the accompanying “BPMN by Example” document exceed cognitively manageable limits (Moody 2009).

Conceptual integration refers to the overview modelling capability for a visual language. Kim described it as the one of the most important mechanisms for any visual modelling languages, which provide a “big picture” view of the domain being modelled (Kim et al. 2000). It acts as an overall cognitive map into which information from individual diagrams can be assembled. Many existing process and service modelling languages, including BPMN, lack such overview presentations of services and service organisation.

Multi-view tool support has been applied in many such systems to mitigate this problem but this increases hidden dependencies and requires long-term memory to retain the mental mappings between views. There are many commercial tools available for business process modelling and simulation. However, despite their increasingly sophisticated functionalities, there are still obstacles to their widespread use (Ali 2009; Baeyens 2007). A common issue is a conflict between usability and flexibility. Typically, the more flexible functionality a tool intends to provide, the more difficult the tool will be to use (Anderson and Apperley 1990; Baker 2002).

Earlier work (Anderson and Apperley 1990; Phillips 1995; Li and Phillips et al 2004) on modelling complex user interfaces and their behaviour, based on the Lean Cuisine+ approach, demonstrated that a tree-based overlay structure can effectively mitigate complexity problems such as those noted above. Lean Cuisine+ (Phillips 1994a; Li and Phillips et al 2004) is a graphical notation based on the use of tree diagrams to describe systems of menus. A menu is viewed as a set of selectable representations (called *menemes*) of objects, actions, states, parameters and so on, which may be logically related or constrained. It uses an overlay structure for specifying the underlying behaviour of event-based direct manipulation interfaces (Phillips 1994b). The success of the Lean Cuisine+ approach motivated us to adopt a tree and overlay approach to mitigate the common complexity issue and cobweb/ labyrinth problems in current business modelling notations and to fully address the modelling requirements summarised above.

From our motivating example, informal discussion with some target end users, and associated literature review, we identified the following key research questions we wanted to answer in this research:

- RQ1: can a tree-and-overlay based service modeling approach provide advantages over existing box-and-line based visual business process modeling languages?
- RQ2: can an effective and efficient environment be implemented for modeling, checking and generating service orchestrations from these tree/overlay-based models?

3. Our Approach

To answer these research questions we initially informally interviewed several target end-users from our University's service provision team, several industry contacts working in the BPM and automated ERP system generation spaces, and several experienced developers of web service-based systems who were users and composers of services, vs service developers. In addition, we identified the following set of requirements for a tree-and-overlay based service modeling and support tool:

- a) Non-programmer end users, such as business process domain experts, need to be able to efficiently model distributed complex systems and related collaborations from a set of pre-existing services;
- b) Users need to be provided with multi-level abstractions to assist in defining different service-based process specifications;
- c) Automated checking of service composition specifications within the tool is needed, using appropriate formalisms and feedback;
- d) The notation must be integrated effectively with other modelling approaches e.g. BPMN;
- e) The tool must provide automatic generation from visual models to industry standard code e.g. BPEL scripts;
- f) The tool must provide large scale diagram support and interactive visualisation and debugging of generated service compositions

We then iteratively designed, developed and evaluated the EML and its supporting environment MaramaEML. We identified our target end users as business process modelling domain experts and service composition experts that have some IT knowledge, but not necessary very detailed programming and service implementation knowledge. However, as with other process modelling languages such as BPMN, very technical end users, such as service implementers, are envisaged to also be able to use the approach. EML uses the concept of a hierarchy of services and service categories to organise a large number of organisational services (e.g. see Figure 2). It uses the concept of multiple overlays over this tree to define business processes that compose and orchestrate these constituent services (e.g. see Figure 8).

The iterative design/evaluation approach we employed comprised two methods: 1) use of the Cognitive Dimensions framework (Green and Petre 1996) to inform our design and to iteratively improve it and 2) use of a formative end user evaluation once a preliminary toolset had been developed to identify weaknesses that required improvement. In both cases the results were useful in refining the language and toolset design. We elaborate on these design methods below

3.1 Initial EML and Tool Design using the Cognitive Dimensions Framework

The CD approach (Green and Petre 1996) is a popular, psychologically based, heuristic framework designed for quickly and easily evaluating a visual language environment. It sets out a small vocabulary of terms designed to capture the cognitively relevant aspects of structures, and shows how they can be traded off against each other. We used the CD approach to help design EML and to decide on features needed in its support tool. The three authors were the experts who carried out this evaluation.

We initially defined a set of prototype EML constructs and associated tool designs from the set of initial requirements we identified as above. We then used various dimensions from the CD framework to assess their characteristics. We took each key EML construct and supporting tool feature and assessed them in turn. We made a number of trade-offs between different dimensions as we refined the notation and tool support features. Two examples of the way in which we used specific CDs for refinement are:

Abstraction Gradient (types and availability of abstraction mechanisms). To reduce the abstraction gradient, we decided to use a tree layout as the basis for all service modelling situations (other than the integrated BPMN modeller). We reasoned that a solid tree layout would be easy to understand and thus provide minimal abstraction gradient for our target EML end-users.

Diffuseness (verbosity of language). Our original EML design was quite verbose, including a variety of notations and formalisms. After our initial CD analysis we reduced the number of notational elements significantly, and eliminated a form based metaphor included in the original design. The tree overlay became the dominant metaphor. It uses a relatively terse set of language elements and hence is easy to learn. Further details of this preliminary CD-based design work can be found in (Li, 2010).

3.2 Formative User Feedback

After we completed an initial version of MaramaEML, we invited several Computer Science and Software Engineering students to carry out a task-based formative evaluation of the refined EML and the MaramaEML prototype. Our objective was to assess how easy it is to learn to use EML and its support tool and how efficiently it could solve the diagram complexity problem. This was a qualitative evaluation and the results were

used to refine the tool's design. Although this version of MaramaEML was functionally near fully featured, the icon definitions were quite simple and it lacked some user interface "niceties". However, the results of the evaluation were promising.

Method

The main functions of the version of MaramaEML evaluated were:

- EML service tree and process overlay modelling ability.
- Basic BPMN modelling ability (to both compare it to EML but also ensure support for BPMN itself)
- BPEL code generation from process overlays

EML and MaramaEML were briefly introduced to the participants who were then asked to perform several predefined modelling tasks. The tasks were divided into three difficulty levels: simple, medium and complex. Participants were asked to repeat the same task in two different environments (pen and paper based EML modelling and software tool-based integrated EML and BPMN modelling). In the pen and paper tasks, users were asked to use a black pen to draw the basic tree structure, a green pen to add the process overlay, and blue and red pens to represent task and trigger overlays on top of the tree (details of these are in the following section). The process was observed and each participant was interviewed at the end of the evaluation. The qualitative feedback was then analysed and key limitations and possible solutions identified.

Results

Six Computer Science and Software Engineering students used the tool and carried out these tasks. All six users completed all the tasks. For the simple modelling task, three out of six users found the pen and paper based modelling approach was more efficient than using the software modelling tool. However, for medium and high complexity tasks, all the users found the software tool was better than pen and paper.

One strong finding was that EML and MaramaEML were very straightforward to use and understand. Users found the tree overlay method reduced the complexity of business processes compared to using only conventional BPMN views. They also found the multi-view support to be a useful approach to enhance the modelling strength.

Several limitations of both EML and MaramaEML and several potential improvements were identified. These included a need:

- for a more detailed mapping traceability between EML and BPMN views
- to provide an integrated environment for the two modelling languages (EML and BPMN)
- to improve visual quality of both EML and BPMN views
- to verify generated BPEL code to guarantee its execution quality
- to enhance diagram scalability (coping with complex and large modelling diagrams)

3.3 Refinement

Based on the formative evaluation feedback, we developed a revised version of MaramaEML, which addressed the identified limitations. This including, for example, addition of the following features:

- A text based log file was created to enhance system traceability
- All modelling views (EML and BPMN) were integrated into the same development environment
- A Labelled Transition System Analyser (LTSA) engine was integrated to verify generated BPEL code. If the code passes the validation, an extra LTSA graphical view of the system is constructed
- New icons and layouts were developed to improve the visualisation quality of EML and BPMN
- Zooming and fisheye view functions were added to enhance visualisation scalability

These and other features of both EML and MaramaEML are described in detail in the following two sections.

4. EML

EML models primarily use a novel tree overlay structure: complex business systems are represented as service trees and business processes are modelled as process overlay sequences on the service trees. In this section we describe the core service tree representation, the overlay mechanisms, including how to represent exceptions, dependencies and other enhancements to the basic overlay approach.

3.1 Service Tree Structure

EML uses a tree layout to represent the basic structure of a service. We chose to use trees as they are familiar abstractions for managing complex hierarchical data for business modellers and business people; they can be easily collapsed and expanded to provide scalability; they can be rapidly navigated; and they can be overlaid by cross-cutting flows and representations of concerns. All the Services, sub-Services and Operations are organized in a hierarchical tree structure to model the system. Connections between the three types of component reflect the functional relationships between them. This provides a service taxonomy that we have found very effective for large numbers of services. Depending on the technology used to realise the services, this can also be used to organise both sub-groups of operations and services into various different groups. Typically the same end users develop the service tree, or sub-trees, as those who develop process flows. However, different end users may be responsible for each.

The basic modelling rules are:

- An Enterprise system must have at least one Service Tree.
- Every service tree must have only one Service node. It may (or may not) include an arbitrary number of sub-Service nodes.
- A Service node is always at the top of the single service tree structure. It must include at least one Operation node (directly or indirectly). It may include an arbitrary (possibly zero) number of sub-Services.
- A sub-Service is contained inside a Service or sub-Service node. It must include at least one Operation (directly or indirectly) and may have an arbitrary (possibly zero) number of sub-Services.
- An Operation is the leaf node of the service tree that is used to carry out a discrete task. It cannot include any Service, sub-Service or other Operation.

A Service is a configuration of technology designed for organizational networks to deliver which satisfies the needs, wants, or aspirations of customers (Feng and Lee 2010; Hanna 2002; Hudak 1989). In EML, a *Service* is a compound operation group that is defined by a list of other activities (sub-services or operations). A *Sub-service* is a graphical object within a service tree, but it also can be “opened up” to show another sub-service grouping. Services and sub-Services share the same shape notation in EML, a small circle. The user can pre-define different colours to distinguish different groups of services / sub-services. All the services and sub-services also have an open centre so that pre-defined EML enhancement function icons can be included within the shape to help identify extra functions (e.g. Elision, Reuse etc.). The name of the service / sub-service is placed outside the circle boundary and positioned arbitrarily around the notation (normally at the bottom or right side of the service / sub-service node). In EML a service has five different execution states, they are:

- *Un-executed / Skipped* (the default state, when the service is not invoked or is skipped)
- *Finished* (the service has completed successfully)
- *Failed* (errors were detected when the service was last executed)
- *Aborted* (the service is killed in the middle of execution)
- *Other* (other unexpected states)

An Operation is an atomic activity that is included within a Service. An Operation is used when the function in the Service is broken down to a finer level of Process Model detail. Operations are the leaf nodes of the Service tree. A square shape (with orthogonal corners) represents an atomic operation inside a service (the operation and service are connected by a tree branch). The user can use different fill colours in operations to distinguish different operation groups; a light grey is used by default. The Operations also have an open centre so that EML enhancement function icons can be included within the shape to integrate other functions (e.g. Exception Handler). The name of the Operation is placed outside the rectangle boundary positioned arbitrarily around the notation (normally at the bottom or right side of the node). A normal Operation square is drawn with a single thin black line. But in certain circumstances (e.g. Single Loop Operation, in Process overlay, in Dependency Trigger etc.), EML changes the boundary style to represent additional information.

Figure 2 shows a complex, fully expanded overview of an EML tree modelling a Travel Planner service. A Travel Planner is a web service-based enterprise system to help users to organize trips. Customers use the client application to submit itinerary enquiries to a travel agent. The agent service receives the requests and communicates to travel providers (Airline, Hotel etc.) to find out a suitable booking.

- A *Travel Planner* is a Service node in the tree. It has three sub-Services (*Customer Service*, *Agent Service* and *Provider Service*).
- There are six Operations inside the *Customer* sub-Service (*Send Book Request*, *Consider Itineraries*, *Send Confirm Information*, *Make Payment*, *Cancel Booking* and *Receive Invoice*).
- The *Agent* sub-Service includes another three sub-Services (*Prepare Itineraries*, *Payment Control* and *Product Booking*) and two Operations (*E-mail Out* and *Print*). There are also lists of different Operations or

sub-Services in the above three sub-Services.

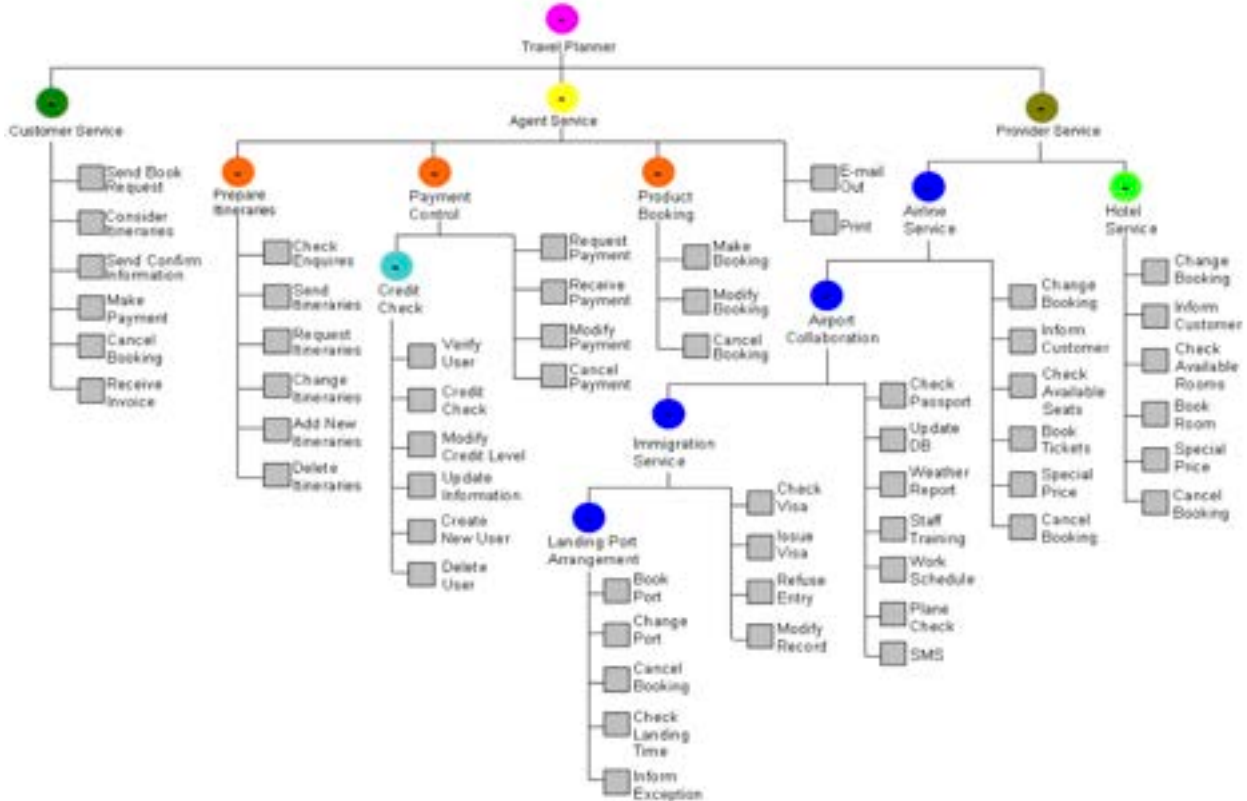


Figure 2: Example EML Tree Structure

The *Provider* sub-Service has a multi-level hierarchical sub-Service tree structure. It has *Airline* and *Hotel* sub-Services at the first level, and the *Airport Collaboration* sub-Service is structured under the *Airline* sub-Service (level two). The *Immigration* sub-Service is inside the *Airport Collaboration* sub-Service at level three. It also includes another bottom level sub-Service *Landing Port Arrangement*. There are twenty-eight Operations involved in this multi-level sub-Service tree.

3.2 Elision

In order to manage diagram complexity, symbols inside each service identify the elision level of the service visualisation. With a tree-based visualisation, a collapse/expand elision mechanism is a natural way to provide complexity management. Most users are familiar with such an approach due to its commonality in graphical user interfaces and desktop user interfaces. In EML a Service or sub-Service can be in a collapsed (elided) mode that hides its details or in an expanded mode that shows its details within the view of the service in which it is contained. The collapsed and expanded forms of the Service / sub-Service objects are distinguished by two markers. A minus (-) symbol indicates all activities in the service have been expanded (Product Booking service node in Figure 4). A plus (+) symbol indicates that part or all of the sub-tasks (services and operations) are elided (Payment Control service node in Figure 4). The elision function only applies to the Services or sub-Services and cannot be used on Operations.

3.3 Service Reuse

EML supports service reuse to reduce structural complexity and to increase modelling efficiency. In EML we chose to represent reusable components using a separate tree. This preserves the overall approach of tree-based decompositions adopted for EML and allows one service tree to reuse elements in another, reusable service tree. The user pre-defines its structure and saves it in a library. Reusable components have a unique ID for future usage. The user can easily attach a reusable component to any branch of an EML tree. The reusable services share the same attributes of Service / sub-Service. But their “Reuse Status” is “True” and “Reuse ID” is a unique number beginning with “R” (e.g. R1). If a reusable service is attached to the EML tree branch, the “Elision Type” attribute will be automatically set as “Collapse”. However, the user can change it to “Expand”.

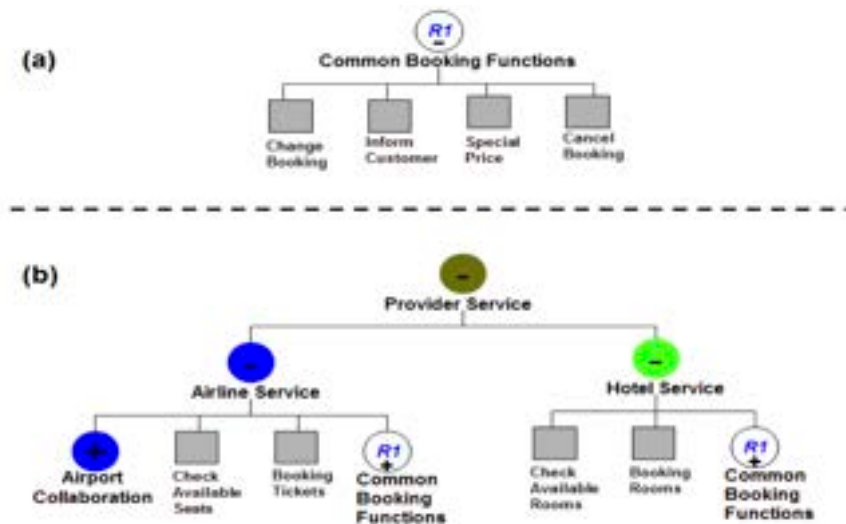


Figure 3(a): Define a Reusable Service; (b): Use Reusable sub-Service in Hotel Service

Figure 3 shows a Reusable Service’s definition and usage. In the *Travel Planner* Service Tree of Figure 2, we see that the *Airline* and *Hotel* sub-Services have several identical operations (*Change Booking*, *Inform Customer*, *Special Price* and *Cancel Booking*). To reduce redundancy, we combine these as a Reusable sub-Service *Common Booking Functions* shown in Figure 3(a). The *Reuse ID* at the centre of the circle is *R1*. Figure 3(b) demonstrates usage of the Reusable Service in the *Hotel* and *Airline* services. The user directly attaches *R1* service node to both tree branches. The elision status of this “*Common Booking Functions*” service has been changed to “*Collapse*” automatically.

3.4 Process Overlay

In EML each business process is represented as an overlay on the basic tree structure or an orchestration between different service trees. In a process layer, users have the choice to display a single process or collaboration of multiple processes. By modelling a business process as an overlay on the service tree, the designer is given a clear overview of both the services structure and the business process simultaneously. Processes can be elided mitigating the cobweb problem commonly existing in flow-based visual notations. In EML, a Business Process may contain more than one separate sub-Process. Each Process may have its own Sub-Processes or share (Reuse) sub-Processes with other Processes. The individual sub-Processes are independent, but could have data connection with others. In order to enhance readability and reduce diagram complexity, EML does not use “Data Flow”. All the data communicated between services, operations and processes are encapsulated in the flows, service nodes and operation nodes. By using this mechanism, the user can focus on the business process itself while complex data details are hidden behind it. However, the user can always obtain (and show) this information from the notation’s data attributes. It provides a more flexible and clearer view to the business processes and service structures.

Figure 4 shows a *Travel Booking Process* in an EML process overlay. Only process related services and operations are shown; other, unrelated services have been elided (e.g. *Payment Control Service*, *Airport Collaboration Service*). The process starts (blue rectangle) with a client side application (not shown) passing a request message to the *Send Book Request* operation of the *Customer Service*. The *Agent Service* receives the request through the *Check Enquires* function, and uses its *Request Itineraries* operation to check availability information with the *Airline* and *Hotel services*. The agent requests flights and rooms with a list of parameters. There are iterations (dashed double arrowheads links) between *Request Itineraries*, *Check Available Seats* and *Check Available Rooms*. When the agent finds that both the air ticket and the hotel room are available on the requested date (end condition C1 & C2), it terminates the loop and sends the client a report generated by the *Send Itineraries* operation. The customer *Considers Itineraries* and *Sends Confirm Information* to the *Agent Service*. The agent receives this information and then *Makes Booking*. After both *Book Tickets* and *Book Room* operations are successfully completed, the agent calls *Make Payment Process* (Figure 6) to ask for the payment and end the existing process (Rounded Rectangle).

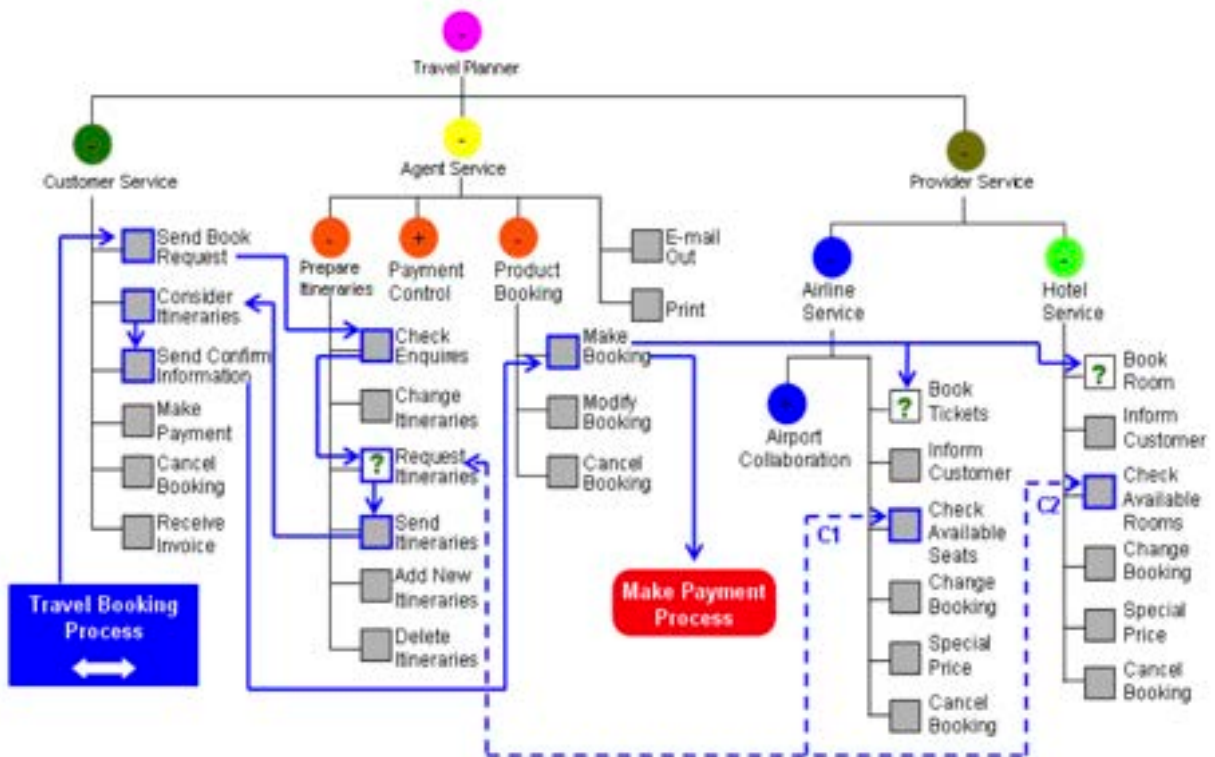


Figure 4: Travel Booking Process Overlay

The double-sided arrow in the Process Start indicates that the process starts with a condition. In this example, it is the arrival of a *Booking Request Message*. This message condition is hidden inside the Process Start notation. The process name inside the Process End notation means that the successful end of this process will lead to the start of a new process (*Make Payment Process*). All the process sequence IDs have been hidden since there is only one process in the diagram. However, the user can make them visible especially when more than one processes appear simultaneously. There are two types of execution state included in this example:

- (1) The finish status (by default). In this state, the boundaries of the operations have been changed to a blue colour. It means that this process step is completed successfully.
- (2) The failed status. In this state, a green question mark appears in the centre of the operation shape (e.g. in *Request Itineraries*, *Book Tickets* and *Book Room* operations). It means that the process step with this operation has not completed normally.

The concept of service “failure” in EML indicates that additional flow(s) are present to handle such situations. The concept of “fail” is used to represent a number of possible situations that require handline, such as an explicit failure return payload e.g. “no rooms available for these days” or empty list/document, invalid or unexpected response from a service, failure to return from service invocation (timeout), etc.

3.5 Exception Handler Overlay

Another type of overlay, an exception handler, specifies what happens if an exception or fail condition changes the normal process flow. A green question mark annotation in the middle of an operation or service indicates an exception handler. Figure 5 shows a hotel room booking exception handler layer. Instead of simply calling *Cancel Booking* and rolling back, we use the EML exception overlay to model a more complete exception solution. If the *Hotel* finds that all standard rooms on the required date have been booked out, it sends a negotiation message (*Change to Luxury Room*) back to the travel agent. The *Agent Service* *Modifies Booking*, *Changes Itineraries* (previous travel plan), makes a new itinerary and sends to the *Customer Service*. The customer receives the latest updated travel plan and *Considers Itineraries*. When the customer makes a final decision, the *Customer Service* then *sends Confirm Information* to the *Agent Service* again. If the user *Accepts* the hotel’s suggestion, the process will lead to *Make Booking* again. Otherwise (*Refuse*), the customer informs the agent to *Cancel Booking* and the *Agent Service* asks the *Hotel Service* to *Cancel Booking*.

The green diamond icon after *Send Confirm Information* is used to represent the conditions. The *Accept* decision is a default option. A dot shape attached at the start of the *Accept* exception flow is used to represent the default attribute. Since the *Refuse* decision is an alternative path, the result (*Cancel Booking*) of this

exception flow remains in the Un-executed status (no border). Meanwhile, the default decision result (*Make Booking*) is in the “Finished” status (green border).

There are another three exception handling icons in *Special Price* and *Request Itineraries* operations and *Airline Service*. The user has chosen to define these in different layers, but they could be combined with *Change to Luxury Room* in the same layer if desired. For a particular tree node, the user can define several different exception layers distinguished by their start conditions. So in EML, exception handling is much more than a simple roll back mechanism. It is an individual process or even a complicated integration of alternative processes. The exception handling overlay is an individual overlay based on the same service tree structure as conventional processes. EML has the freedom to allow the user to define them in a single layer or combine them with the process and trigger overlays described next to generate an integrated overview of the system.

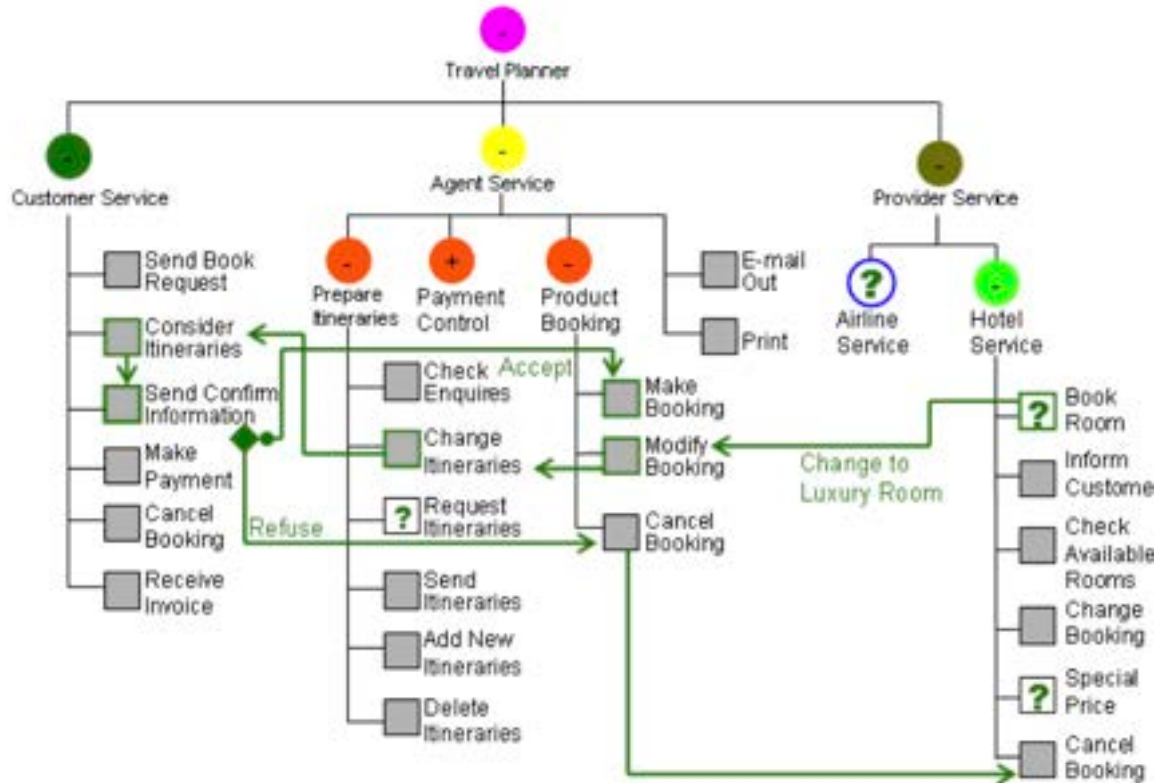


Figure 5: Hotel Room Booking Exception Handler Overlay

3.6 Dependency / Trigger Overlay

An EML trigger overlay is used to model internal system dependencies. Since EML uses a multi-layer structure, users can choose to combine trigger layers with process layers or separate them by using different views to reduce diagram complexity. The major difference between a process and a trigger is the actor involved. A process is performed by a user; the sequence and result of a process are normally variable. A trigger is enacted automatically by the system itself. As it is an internal system dependency, the trigger execution order and outcomes (for the same trigger condition) are usually unalterable. Thus another benefit of the trigger overlay is to support internal dependency process reuse. Once a trigger is defined, it can be reused in different processes and exception handlers.

Figure 6 illustrates the *Make Payment Process* example with trigger flows. It follows the *Travel Booking Process* example from Figure 4. When the user successfully books air tickets and hotel rooms from *Travel Booking Process*, the *Agent Service* starts to *Request Payment* using *Payment Control* service. The agent sends the payment request to *Customer Service*, and the customer then *Makes Payment*. If the agent *Receives Payment* within three days (default option), then it sends the invoice by *E-mail* to the customer. The customer *Receives the Invoice* and the whole process ends. However, if the agent *Payment Control* service doesn't receive the payment in three days, it *Cancel the Booking* using *Product Booking* service.

This operation triggers two extra operations automatically: *Cancel Booking* in *Airline Service* (T1.1) and *Hotel Service* (T1.2). A “triggered” operation is used to indicate additional, cascading operations, often run in parallel, augmenting the process much like an “extends” relationship in use case modelling. Triggered operations can become large additional flows and we use this construct to help manage flow complexity in EML overlays. In this example, we integrate the trigger with the process overlay. However, the user can hide the

trigger and define it in a different layer to reduce diagram complexity. The *Cancel Booking* trigger itself is reusable. For any other processes an exception handler or trigger can be directly reused by linking the flow to the operation and filling in a trigger start condition.

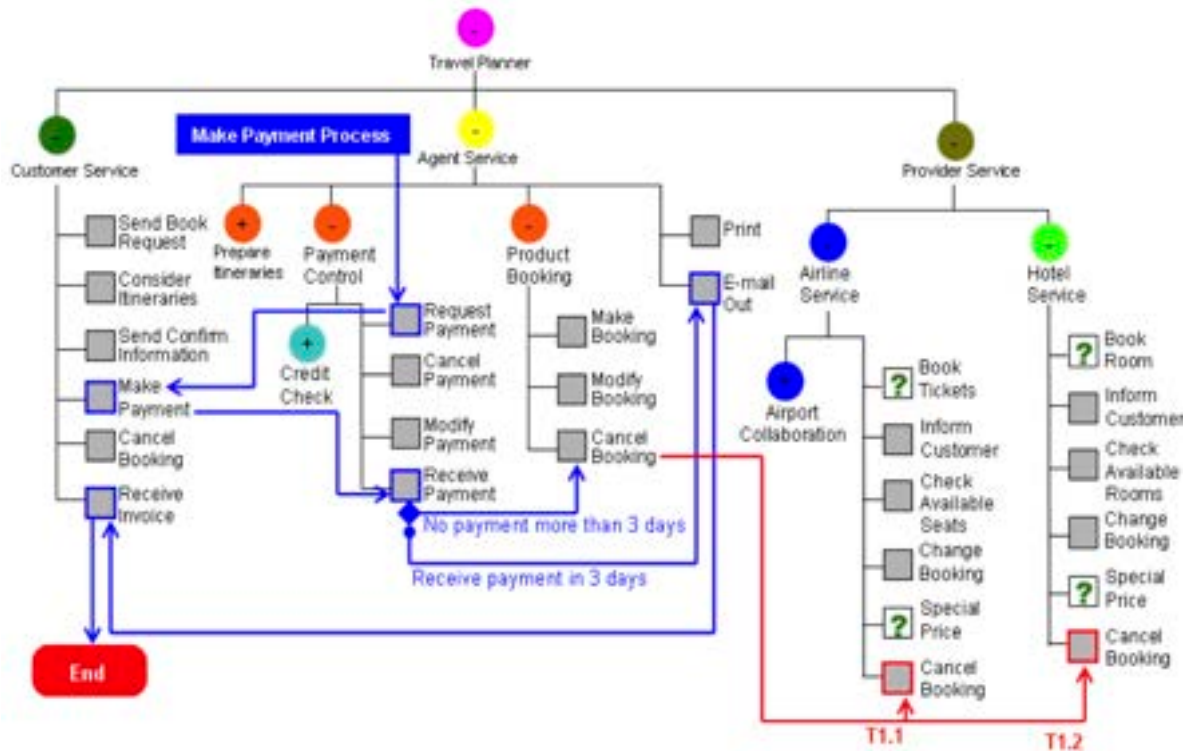


Figure 6: Make Payment Process with Triggers

3.7 Iteration

We represent iterations occurring in different overlays by using the same visual method (styled lines), but use process specific colours to distinguish them. This increases the consistency of the notation and reduces the modelling complexity of the diagram. There are three types of loops:

1. A single activity loop is represented as a single arrowhead dashed line whose source and target are same operation (or service/sub-service). Attributes in the dashed flow control the iteration (e.g. loop times, start and complete conditions, input/output data etc.).
2. Loops of two operations (or services / sub-services), use a dashed line with two arrowheads. Figure 4 includes the iteration between *Request Itineraries*, *Check Available Seats* and *Check Available Rooms* operations in the trigger overlay. When the *Agent Service* receives the room and flight ticket booking application from the *Customer Service*, they need to check whether the suitable hotel room and flight ticket are available on a customer required date. The customer normally sends a list of preferred date, room and ticket types for the room booking. The *Agent Service* starts from the highest preference date, room and ticket type to the lowest preference types. The process loops until the termination conditions *C1* (finds the suitable flight on required date or there is no flight available on all the preferred dates) and *C2* are met (finds the suitable room on required date or there is no suitable room available on all the preferred dates).
3. If a loop involves more than two operations (or services / sub-services), a single arrowhead dashed line guides direction, linking different operations or services in a closed circuit.

3.8 Conditions

A condition shape is a diamond. The fill colour of the condition is based on the overlays. If a shape is used in a process layer, it appears in blue (in Figure 6). However, if it is in an exception handler or trigger overlay, then the colour becomes green (in Figure 5) or red. All conditions icons are filled to indicate conventional exclusive if/then/else type semantics, or may have an open centre indicating other conditional semantics e.g. OR, AND, XOR or OTHERS.

4. MaramaEML

The MaramaEML environment allows designers to model with EML and generate Business Process Execution Language (BPEL) orchestrations of web services from EML models (Li et al 2008). MaramaEML is implemented using our Marama meta-tool (Grundy et al 2008; Grundy et al 2013) as a set of Eclipse plug-ins, providing a robust, scalable design tool. It integrates EML and BPMN modelling with BPEL code generation and LTSA checking. Performance simulation (Grundy et al 2006) is also incorporated in the integrated EML support environment, facilitating cost-effective tests of the integrated specifications using random data and visualisation of test results using the same design-level specification views. We use the university enrolment system example of Section 2 to illustrate MaramaEML's major functionalities and also the application of EML to another substantial example.

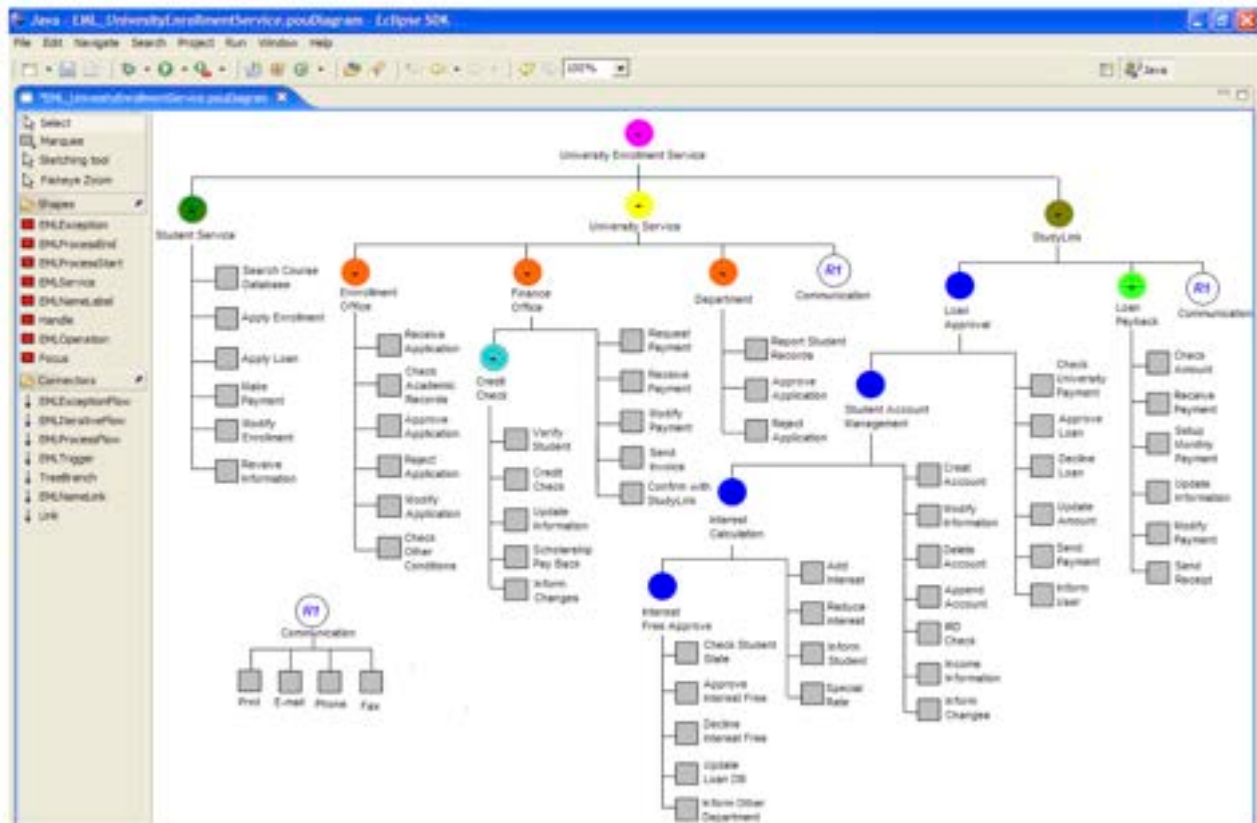


Figure 7: University Enrolment System Overall Structure

4.1 Service Tree Modelling

Figure 7 shows a complex, fully-expanded overview of an EML tree modelling the university enrolment service using MaramaEML. The student service, university service, and StudyLink are sub-services (represented as ovals) of the university enrolment service. The university service includes five service groupings (enrolment office, finance office, credit check, department and communication). The rectangle shapes represent atomic operations inside the service. The StudyLink service also includes a detailed four layer sub-service structure. MaramaEML supports EML's service reuse notation. In Figure 7, the *Communication Service*, defined as a reusable component at bottom left), is reused by the *University Service* and *StudyLink Service*.

4.2 Overlay for Processes, Exceptions and Triggers

In EML each business process is represented as an overlay on the basic tree structure or an orchestration between different service trees. For example P1.1 to P1.17 in Figure 8 shows the *Enrol in a Course* process on the University Enrolment Service tree. The process starts with a process name followed by a process flow (blue arrow) representing the sequence. Each flow has a sequence number; for a complex process, users can use this to model concurrency / synchronization. Involved operations or services have bold outline borders to help identify the track. Data is bound to a process flow to flow in or out of operations. In this process, the student uses *Search Course DB* to select the suitable course and *Applies Enrolment*. The enrolment officer *Receives Application* and *checks this student's Academic Records* with the *Department*. As soon as the *Department*

Reports the student's record and Approves the course Application, the enrolment officer will Check other Related Conditions and ask the finance officer to Request the Payment. The student then Applies Loan and StudyLink Checks University Payment information with the Finance Office and decides if it Approves or Declines the Loan. If the university receives a payment from StudyLink, the finance officer confirms the enrolment and Sends the Invoice to the student.

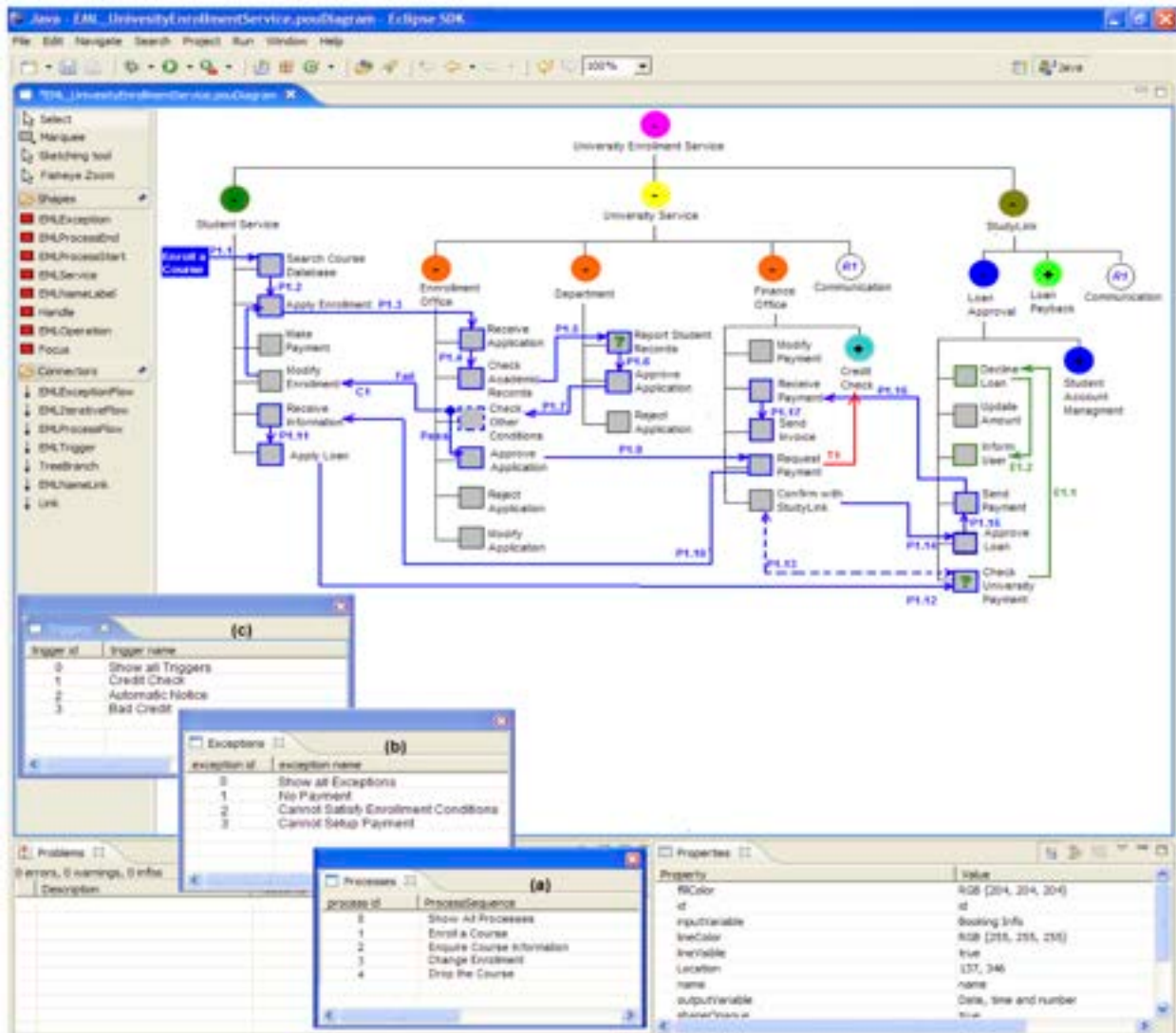


Figure 8: Using EML Overlays to Model the Enrol in a Course Process

In a process layer, users have the choice to display a single process or collaboration of multiple processes. In MaramaEML, a user can selectively show/hide EML Processes (Figure 8 (a)), Exceptions (Figure 8 (b)), or Triggers (Figure 8 (c)). Note also the user has elided parts of the tree to allow focus on branches that are relevant to the process displayed. T1 in Figure 8 shows a trigger condition. Here, when the Finance Office Requests Payment from the student, they also need to do a Credit Check. MaramaEML users can define trigger conditions as attributes, via property sheets, at each end of the connector to control the dependency. Users can choose to combine triggers with the process layers (as in this example) or separate them, using different views to reduce complexity. Figure 8 also includes two overlaid exception handlers for handling course prerequisite failure (in Report Student Records) and failure to confirm loan application in Check University Payment. Conditions for these are also specified in property sheets, as are the conditions for conditional flows, such as the one attached to the boundary of Check Other Condition.

4.3 BPMN Integration

MaramaEML allows other business process modelling notations to be integrated to collaborate with EML and to facilitate modelling of different structural and behavioural aspects. For instance, in EML, data are bound to process flows via textual properties so as to reduce diagram complexity. However, sometimes a user may require this kind of information to be presented directly in the diagram. BPMN diagrams can represent the

internal flow sequence of data well, but this kind of flow-based approach can easily cause diagram cobweb problems. An ideal solution is to provide the user with access to both diagram types. MaramaEML includes linked BPMN and EML views with consistency management between them i.e. when an item is changed in the EML view, the corresponding BPMN item(s) are changed and vice-versa.

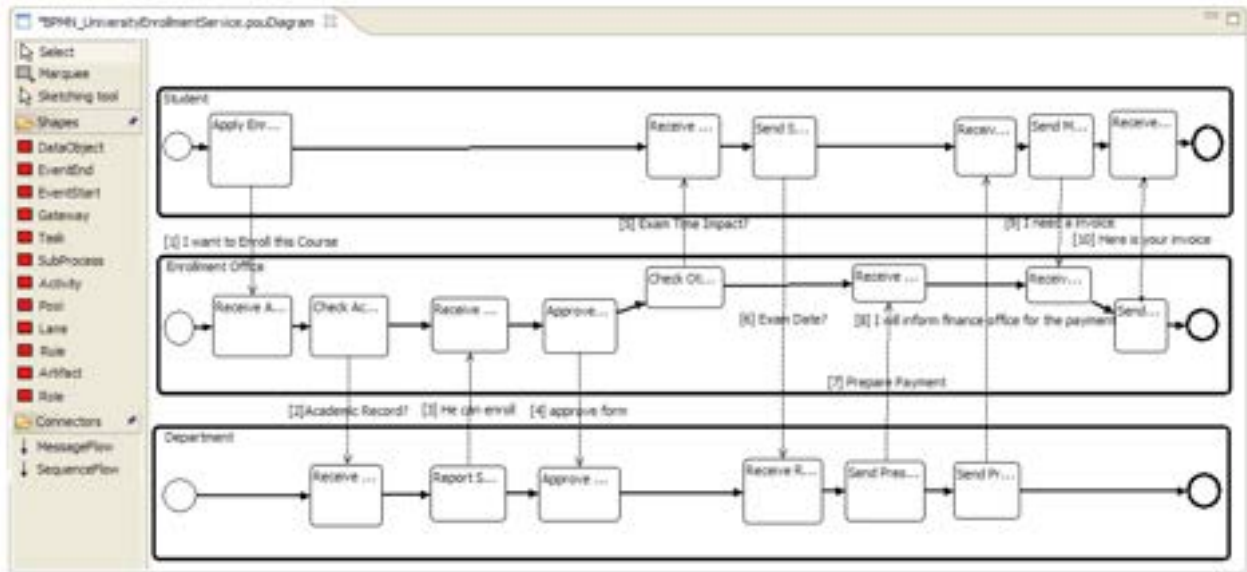


Figure 9: BPMN View --- Enrol in a Course

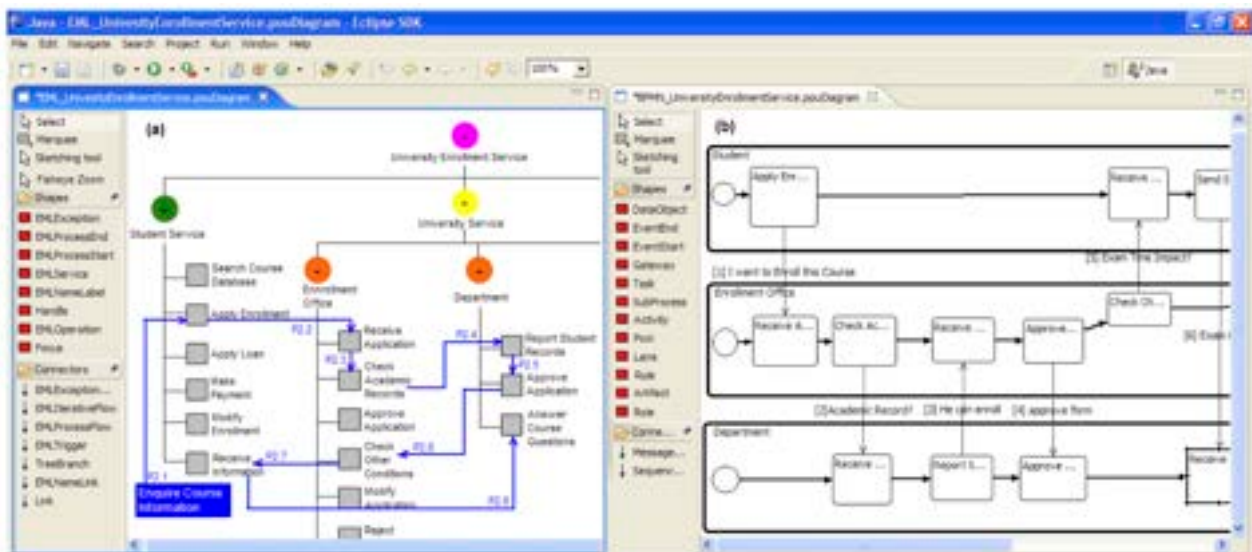


Figure 10: Using EML and BPMN views to model the same process

Figure 9 shows a BPMN view for the “Enrol a Course” process. The *Student*, *Enrolment Office* and *Department* are described in three pools. Figure 10 shows the juxtaposition of the EML view (a) and the BPMN view (b) to model this process. From the EML view the user can obtain a clearer service structural view and an understanding of the process sequence, while from the BPMN view, the user can also see the data transformation. These different domain-specific visual language process notations share a canonical merged model in the MaramaEML tool. Modifications made by the modeller to either notation are reflected in the other.

4.4 BPEL Generation and LTSA Validation

MaramaEML can generate Business Process Execution Language (BPEL) code and coordinate processes in a BPEL workflow engine, allowing users to export and integrate EML process specifications with other BPEL compatible environments. We have also integrated an LTSA-based model checker (Foster and Magee et al 2003) into MaramaEML to verify the correctness of the EML models. As shown in Figure 11, the EML process layer (a) has been automatically compiled to executable BPEL code (b). Our code generator performs model

dependency analysis and maps EML model constructs to structured BPEL activity constructs. The LTSA engine then verifies the correctness of the generated BPEL code. To generate the BPEL code, the user needs to move to the EML tree structure view (a) and use a popup menu (e) to call the “Generate BPEL4WS from EML” function. BPEL code will be automatically compiled and displayed in area (b). To verify the BPEL code, the user needs to change to the “LTSA Perspective” from (h), open the target BPEL code in (b), select the process name (f) and choose an LTS compile function from popup (g). The output from validation appears in (d) and the final LTS view appears in (g). If there is no compilation error, a LTS diagram is presented (c). Errors checked for include incorrect sequencing of web service orchestrations, incorrect parameters, orphaned process stages, non-terminating processes and unreachable processes.

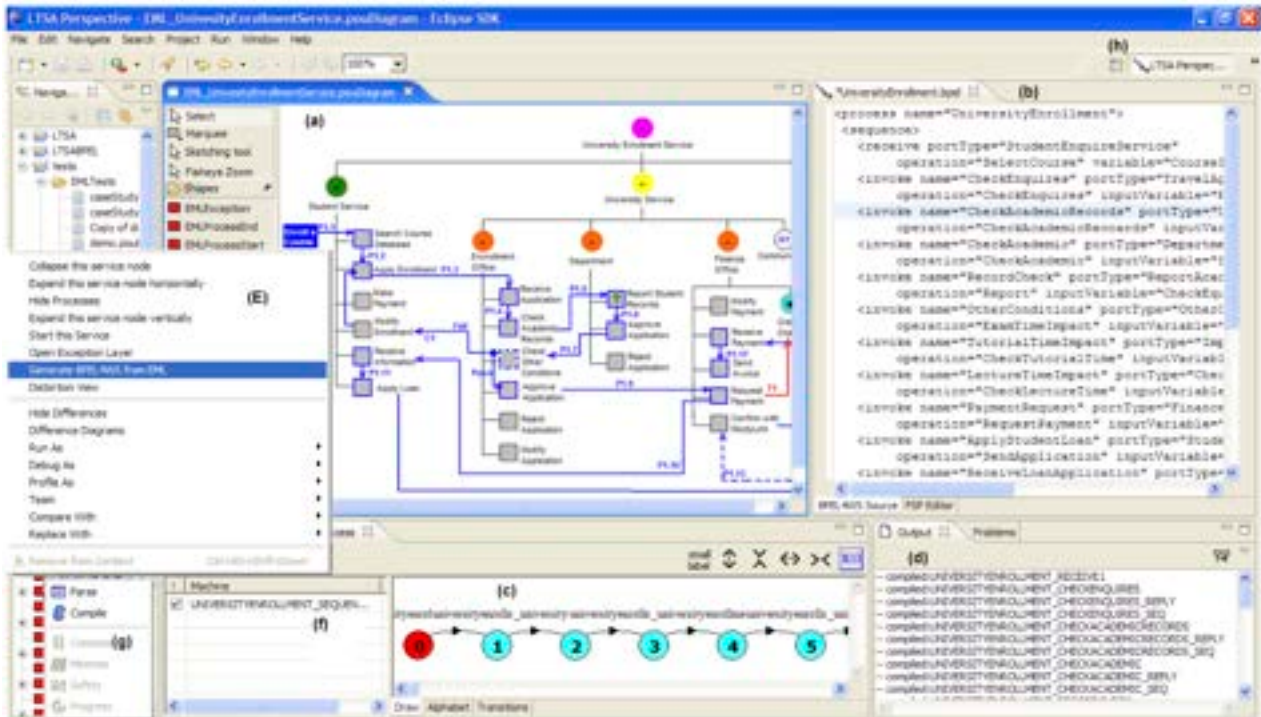


Figure 11: BPEL Generation and LTSA Code Validation

4.5 Zoomable and Fisheye Views

EML’s novel tree overlay structure has reduced the modelling complexity at a visual methodological level. However, due to the nature of enterprise complexity, sometimes views can still be very large. At a technical level, to enhance EML’s diagram navigability and understandability a zooming (radar view function) and a distortion-based fisheye zooming function have been developed in MaramaEML.

Figure 12 shows a MaramaEML zooming view (a). The user draws a “Radar square” (blue square) in the tree overview area (b). As the user moves the blue radar square, the components in area (a) are panned to focus accordingly. By using this function, the user can have an overview of the whole tree structure, and also be able to navigate to detailed parts. Figure 13 shows a MaramaEML fisheye view (a). The user draws a “fisheye area” (blue square) in area (b). Components in the blue square are represented in area (a) at enhanced size (*Department* Sub-tree), while the rest is distorted with the degree of shrinkage increasing with the distance from the fisheye area. As the user moves the blue radar square, the components in area (a) are moved accordingly into focus. In the example, the starting shrinkage degree is 2, which is much larger than desirable, but serves to illustrate the mechanism. At any stage, the user can change this value by selecting from a pull down menu.

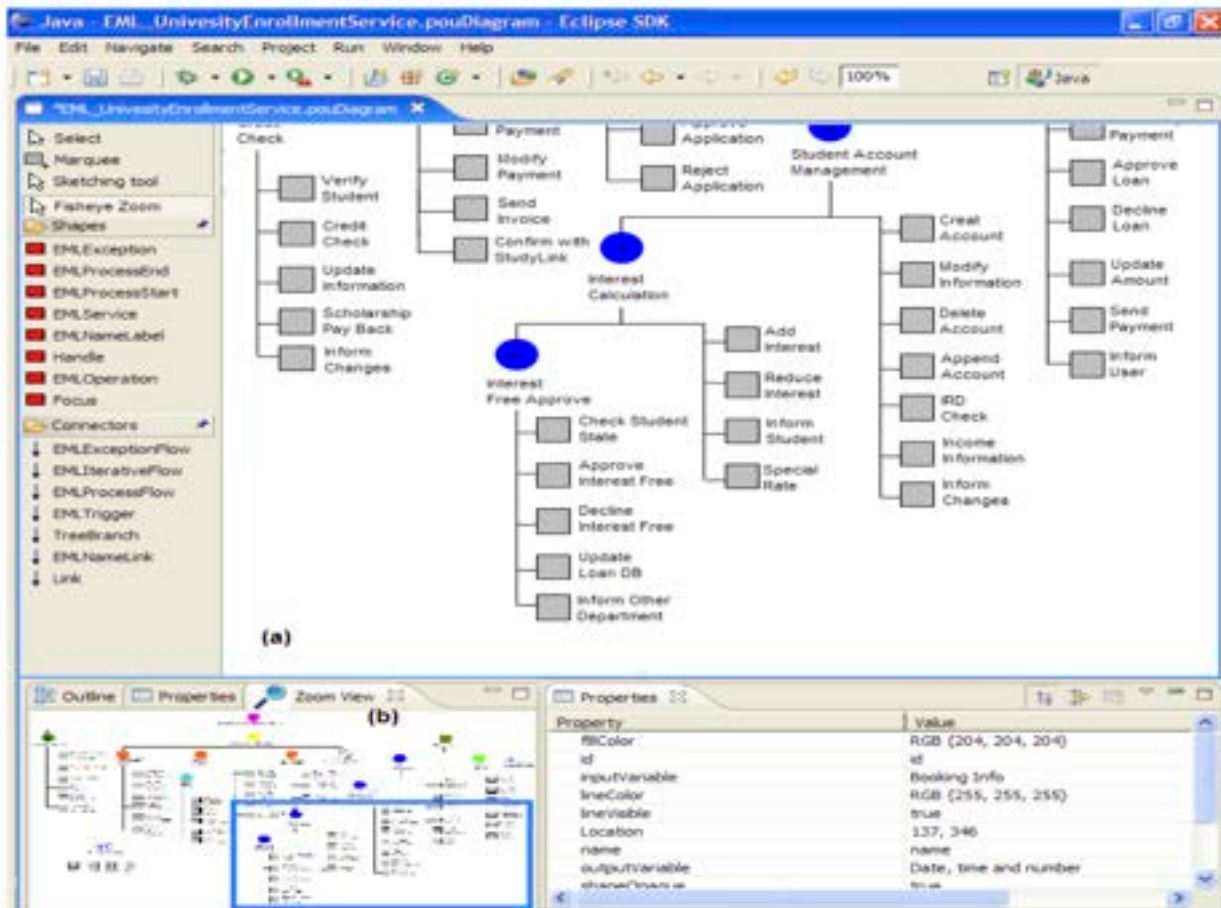


Figure 12: Zoomable View in MaramaEML

5. Implementation

MaramaEML was implemented using the Marama meta-tools (Grundy et al 2008, Grundy et al 2013). Structural aspects of MaramaEML were specified visually using Marama’s visual meta-editors. Behavioural aspects were constructed programmatically as Java-based Marama event handlers.

5.1 Service Tree Structure

An event handler for the MaramaEML tree layout was defined for the root/leaf and parent/child shapes to respond to. When a shape is added to a MaramaEML modelling view, the location of the shape is analysed and then corresponding reacting behaviours are executed to automatically layout the shape based on whether the shape is created as a child of a parent shape or is standalone. This event handler catches a *new shape added* event and responds by running a tree layout algorithm. We have developed an algorithm to calculate the vertical and horizontal space between all the nodes in the tree structure. When the user adds a new service or operation node under a service node, the positions of all previous nodes are recalculated and updated to maintain the tree layout; tree branches are rebuilt too. A sub-tree can be moved to a standalone location to become an independent service tree or to a parent shape location to become a migrated child. This is implemented as an event handler to react to a *shape move* event. When a shape is moved, all its subsequent descendants are retrieved and moved together as a whole in reaction to the event.

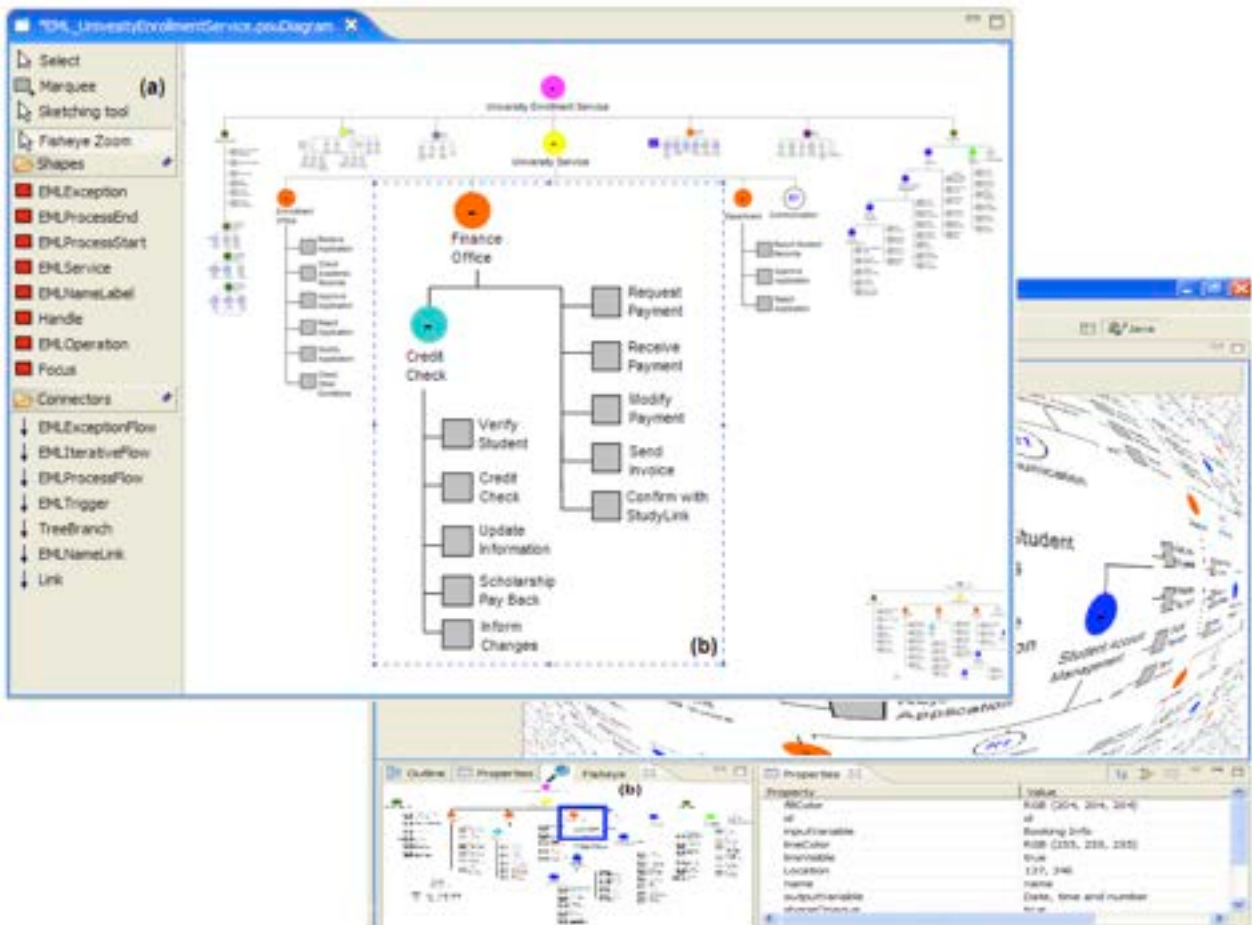


Figure 13: Fisheye Views in MaramaEML

5.2 Overlay

The overlays of process models on a tree structure were firstly defined as normal elements, including the Process Starts, Process Flows and Process Ends as the basis, with Trigger and Exception Flows as complements. Iterations can be defined for each Process/Trigger/Exception Flow via iterative property settings. Multiple process overlays can be modelled in the same diagram, and they are distinguished from one another using unique process identifiers (each process-related element has property identifying which process it is belonging to). Process overlays can be shown or hidden, selected and deleted, to react to the user's interaction. These features were implemented as user triggered event handlers (reacting to right click actions on context menus). In addition, an Eclipse ViewPart implementation called "EML Processes View" provides the user with a more straightforward way (juxtaposed display with the diagram) to view all the processes overlaid in a diagram, while also allowing selection of a particular or a subset of processes to be displayed. In the implementation of process overlays, all the elements of an EML diagram are walked through to identify their notation type and properties. Elements related to process overlays are collected and distinguished using a Hashtable data structure, which is traversed through and analysed to supply on-demand interactive display of multiple process overlays.

5.3 Detailed Properties

In order both to validate trees and overlays, and to generate detailed technical elements of BPEL4WS scripts, many EML constructs have associated properties. These include:

- Service interface names, deployment location and ports, operations, payload types, and some aspects of the Service Level Agreement associated with the service (timeout, alternative location(s), etc).
- Overlay flow sequencing (auto-generated), to and from service location/port, payload details
- Conditional execution and triggered execution properties

Details of these properties and their usage can be found in (Li, 2010).

5.4 BPEL Generation

The BPEL code generation facility is implemented as both embedded runtime behaviour in Marama (by adding to the Marama API) and as a user-triggered event handler. It uses the Marama API calls to query user-defined modelling elements and perform mapping code generation to the file system. The code generation algorithm is straightforward, traversing EML nodes using a Hashtable data structure and analysing the types and properties of the EML elements to permit mapping to the corresponding BPEL code structure. As multiple processes can be defined in one EML diagram via process overlays, multiple BPEL process files can be generated. One-to-one mapping of EML elements to BPEL constructs is typical, for instance, an EML Service maps to a BPEL PortType; an EML Operation maps to a BPEL Operation; an EML ProcessFlow maps to a BPEL Link; an EML ExceptionFlow maps to a BPEL Compensation Handler; an EML TriggerFlow maps to a BPEL Event Handler. The code generator buffers the diagram analysis results, i.e. the XML code snippets contributing to the final BPEL processes definition, and then outputs the completed XML files. Generating complete and executable BPEL code requires additional diagram properties (e.g. input and output data, conditions, error message etc.) to be set via property sheets over and above those needed for basic EML modelling. The generated BPEL is given to the third-party LSTA-based model checker to verify correctness of the BPEL (no orphan states, correct sequencing of web service orchestrations etc) and results of this model checking shown using the LSTA-based model checker's visualisation facilities.

5.5 Zoomable View

Standard zooming functions, including zoom in, zoom out, zoom fit and selection zoom are implemented as toolbar commands on the Eclipse Workbench. To these, we have added an Eclipse PageBookView, which listens to user's diagram selection events, and renders a 'Radar' zoom view for the whole EML diagram. The "zoom" functions zoom in/out the entire EML diagram by a predefined scaling factor. The "zoom fit" function fits all diagram elements in the available screen space with an automatically adjusted scaling factor. The "selection zoom" function allows user to select an area of the diagram and zoom to fit the selected part. The "Radar" zoom view accompanies the EML diagram, providing a thumbnail as well as an indication of visible items inside the screen boundary and those outside of the boundary of the EML diagram. This implementation is integrated with the Marama API. We have provided an additional package inside the MaramaEditor plug-in to manage the zooming functions while still exploiting the existing Marama code base. The package includes zooming interfaces, various zooming actions, Marama diagram mouse trackers, and viewing areas. MaramaEditor is then configured to enable these zooming features using its zoom manager.

5.6 Fisheye View

We implemented a fisheye view function by providing a local context against a global context. This is a focus and context visual technique often referred to as a "distortion based display". Three major attributes control the fisheye function: Focal point, Distance from focus and Degree of Interest (DOI). A point of interest has to be defined which sets the focal point. The "distance from focus" determines the distance from my point of interest (focus) to some point x. Longer distances lead to smaller sizes of the shapes in the distorted display. The implementation of DOI is composed of a static and dynamic component. The static component is either the a priori importance or the global importance of the element relative to every other object in the system. For the user, the global importance is how a tree node is used more than another tree node in the EML diagram. The dynamic component creates a relationship between the user's interest and the importance of an item depending on the latest interactions on the tree. The DOI is assigned to every element in the EML diagram, and a node is selected as the central focus point. It is important to note that if the point of interest changes, then the DOI must be recalculated for every node.

6. Evaluation

As outlined in the Approach section, continuous evaluation has played an important role in the entire EML and MaramaEML design and implementation process using a combination of Cognitive Dimensions based assessments and formative end user evaluations. Recall the research questions we identified from Section 2:

- RQ1: can a tree-and-overlay based service modeling approach provide advantages over existing box-and-line based visual business process modeling languages?
- RQ2: can an effective and efficient environment be implemented for modeling, checking and generating service orchestrations from these tree/overlay-based models?

We have provided preliminary support for an affirmative answer to RQ1 and RQ2 through our formative evaluation. To provide a more summative evaluation addressing these questions, we undertook a formal end user evaluation of MaramaEML in the form of a comparative evaluation. As an example of an existing box and line approach, we chose BPMN and so focused the end user study around a comparative evaluation of EML and BPMN. Combining the comparative aspects of RQ1 and RQ2 leads to two hypotheses, which we tested in our evaluation:

- H1: The task performance of users is better using EML than BPMN
- H2: Users find EML more usable than BPMN

6.1 Method

Our study had a within subjects design to compare the usability and effectiveness of EML and BPMN followed by additional tasks to evaluate the usability of several EML specific features. To avoid a learning effect, in the comparative part of the study all subjects repeated the same tasks for each modelling language, but with half the subjects using EML first, followed by BPMN, while the other half used BPMN followed by EML. A briefing session prior to undertaking the tasks introduced subjects to the two modelling languages and a target example to be modelled. Following completion of all tasks subjects completed a usability survey.

The procedure and tasks undertaken are as follows:

Step 1: Evaluation Schedule Introduction

Step 2: *EML & BPMN Introduction* --- Participants were briefly introduced to EML and BPMN, and some working examples explained (including a Travel Booking system)

Step 3: *nDeva++ Introduction* --- The target modelling example was introduced (nDeva++, a web-based University Enrolment System)

Step 4: Participants download EML Tool, User Guide and EML introduction slides from the website

Step 5: *Modelling Task 1* --- Participants were divided into two groups. Group 1 was asked to model nDeva++ overall structure using EML. Group 2 was asked to model nDeva++ system use BPMN

Step 6: *Modelling Task 2* --- Group 1 Participants were asked to add an “Enrol in a Paper” task process using EML. Group 2 Participants were asked to add an “Enrol in a Paper” task process using BPMN

Step 7: *Modelling Task 3* --- All participants were asked to save their work and use the other language to repeat Step 5 ~ 6 (i.e. Group 1 participants then used BPMN, and Group 2 EML).

Step 8: *Modelling Task 4* --- All participants explored use of the show / hide tasks and trigger functions, show / hide tree component functions.

Step 9: *Modelling Task 5* --- Participants explored use of the fisheye view and zooming functions, code generation and validation functions and other support functions

Step 10: *Answer Questionnaire* --- Participants were required to complete General Usability Questionnaire

6.2 Participants

Thirty-two subjects participated in this evaluation. They were recruited by email invitations sent to industry, academic and graduate student mailing lists. No inducements were offered. Subjects were required to have BPM, computer science or software engineering literacy, resembling the intended range of various end user targets for the toolset. Participants were selected on a “first in” basis from responses to the email invitation. Table 1 shows the range of experience in IT and BPM as self identified by participants. Formal IT indicates tertiary education in Computer Science and/or Software Engineering. Informal or No IT, indicates those informally self trained or untrained in IT. BPM indicates who have had BPM experience – all had been self-taught. Not specified indicates participants failed to self identify in one or other of the categories. From this, we can observe that the participant set is biased

sed towards those with BPM knowledge and also has very limited number of participants without a formal IT training. However, this was somewhat representative of the types of end user we were interested in serving with MaramaEML: business process modelling domain experts and service composition experts that have some IT knowledge, but not necessarily detailed service implementation expertise. We would have liked to have a larger number of non-formally trained IT end users in our sample e.g. with a business management background.

Table 1: Participant background experience in IT and BPM, numbers of participants

	Formal IT	Informal or No IT	Not Specified	Total
BPM	13	5	2	20
No BPM	10	0	0	10
Not Specified	2	0	0	2
Total	25	5	2	32

6.3 Results and Discussion

Each subject took around two hours to complete the briefing, evaluation tasks and survey.

1. Usability of EML

Before addressing the two hypotheses specifically, we first present survey feedback looking at usability characteristics of MaramaEML in isolation to obtain a feel for its areas of strength and weakness. Figure 14 shows averaged results from the 5 point likert scale responses. The “EML Tree Structure Usefulness” received the highest average score (4.8 out of 5). Almost all participants believe that using a tree overlay structure as a modeling approach will strongly enhance their modeling ability. The “overlay usefulness” and “EML General Modeling Ability” received averages of 4.5 and 4 out of 5 respectively. The two least well received features related to MaramaEML’s performance with respect to Scalability and Efficiency. Participants commented on the unresponsive speed of the tool when zooming a large, complex diagram, and the need for better control of information hiding. Looking at differences between the different populations, those with no BPM background on average rated each of the usability characteristics between 0.3 to 0.5 lower than those with BPM experience. These differences were significantly different at a 0.1 level using a t-test, except for efficiency. This would indicate that those with BPM experience saw more value in the abstractions employed in EML than did those without such experience. The small number of participants without formal IT training meant no significant differences could be drawn between those with and without IT training (in this and other characteristics reported below).

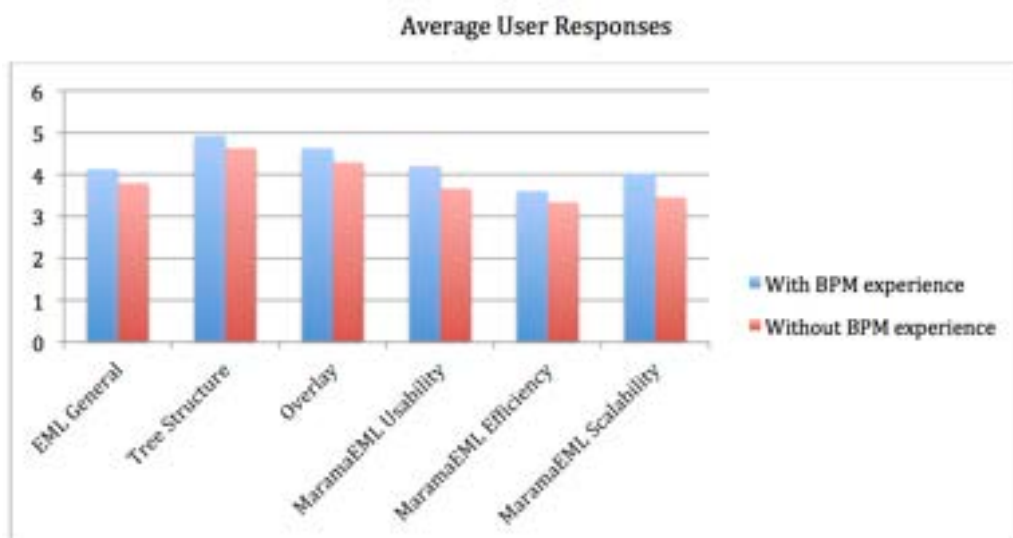


Figure 14: EML and MaramaEML Usability Rating Summary

2. Comparative User Performance

Figure 15 compares the variation in model quality achieved by users when undertaking tasks using EML and BPMN. Only 1 subject could not complete the assigned tasks. To estimate model quality, we collected all answers from the participants and rated them against model answers we had developed. Ratings ranging from

Excellent to Poor were given based on a predefined assessment rubric for each task. These ratings are, by their nature, subjective, but the consistent application of the scoring rubric minimised this subjectivity. Overall, the quality of user EML and BPMN models was very similar. 83% of EML models were rated as good or better, while 75% of BPMN models were rated good or better. However, there was no statistical significance in the differences obtained, when measured using a t-test, at the 0.05 significance level. There was, however, a very significant difference between the performance of those with BPM experience and those without. For both BPMN and EML modelling, those without BPM experience performed very significantly (at the 0.01 level) worse than those with BPM experience, with between 1 to 1.5 scale points difference in the averages. This is unsurprising; one would expect those without BPM experience to produce lower quality models. For those with no BPM experience, their performance with EML was marginally better than with BPMN (at a 0.1 significance level).

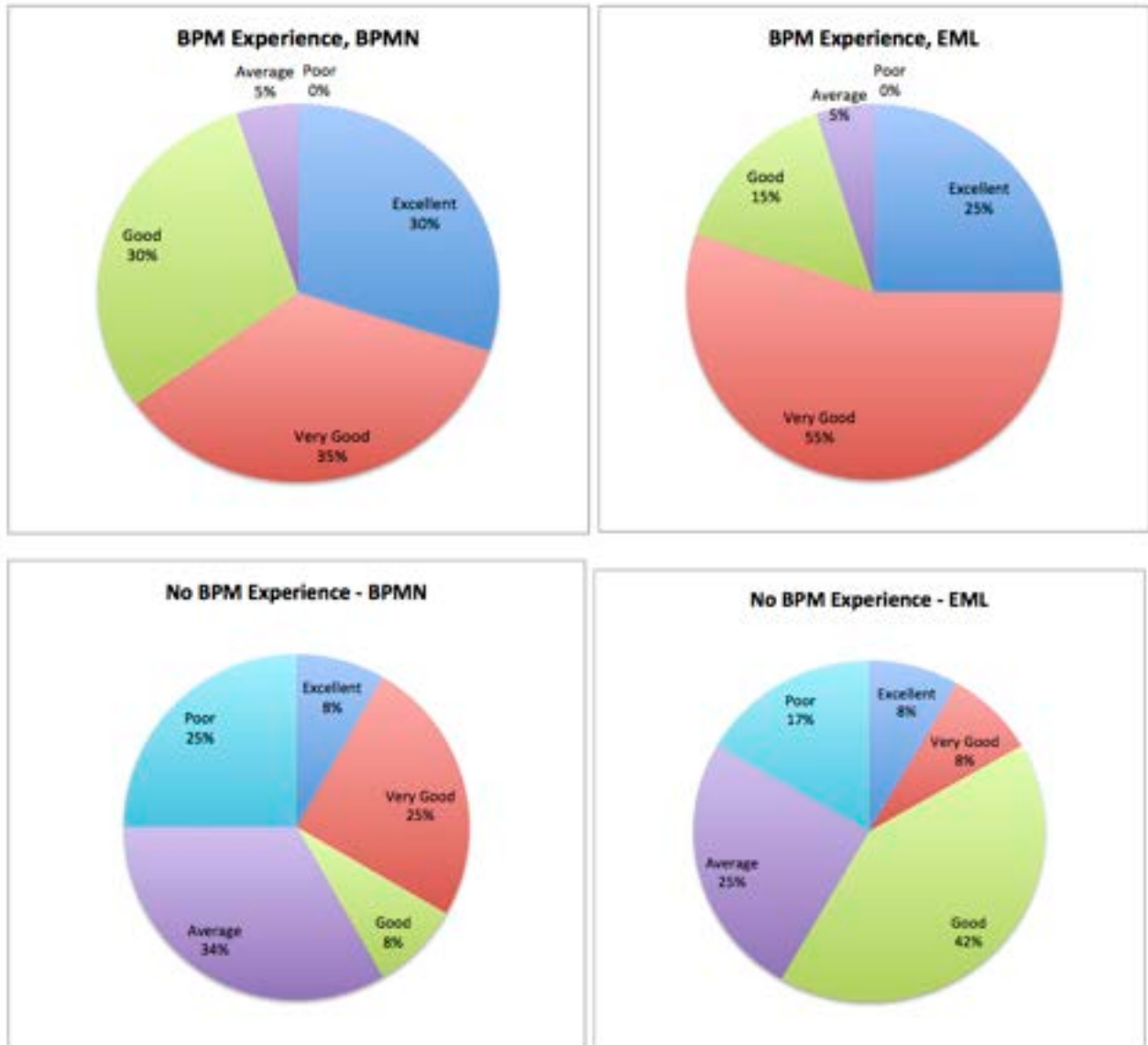


Figure 15: User Performance EML Vs. BPMN

Combined these results support the null hypothesis for H1: i.e. there is no difference in user performance when modelling in EML or BPMN. It is worth noting, however that this result comes from a population where nearly 65% of the total participants already had BPMN experience before performing this evaluation, while EML was a totally new visual language for all of them and they had just 30 minutes training in it at the beginning of the evaluation.

3. Comparative Usability

Figure 16 shows results from our user study of 32 participants rating EML vs BPMN using several usability features introduced in Section 2: Graphical Complexity, Notation Executability, Complexity Management,

Conceptual Integration, Use of Colour and Suitability for Hand-drawing. Each feature was evaluated on a 0 to 5 Likert-type scale, but allowing for users to enter half point values, i.e. effectively a near continuous 11-point scale. For all except graphical complexity, a larger value implies enhanced usability. In all cases the differences in the mean values, as measured using a t-test, are significant at the 0.01 level, making them highly significant. We discuss these results in the subsections below.

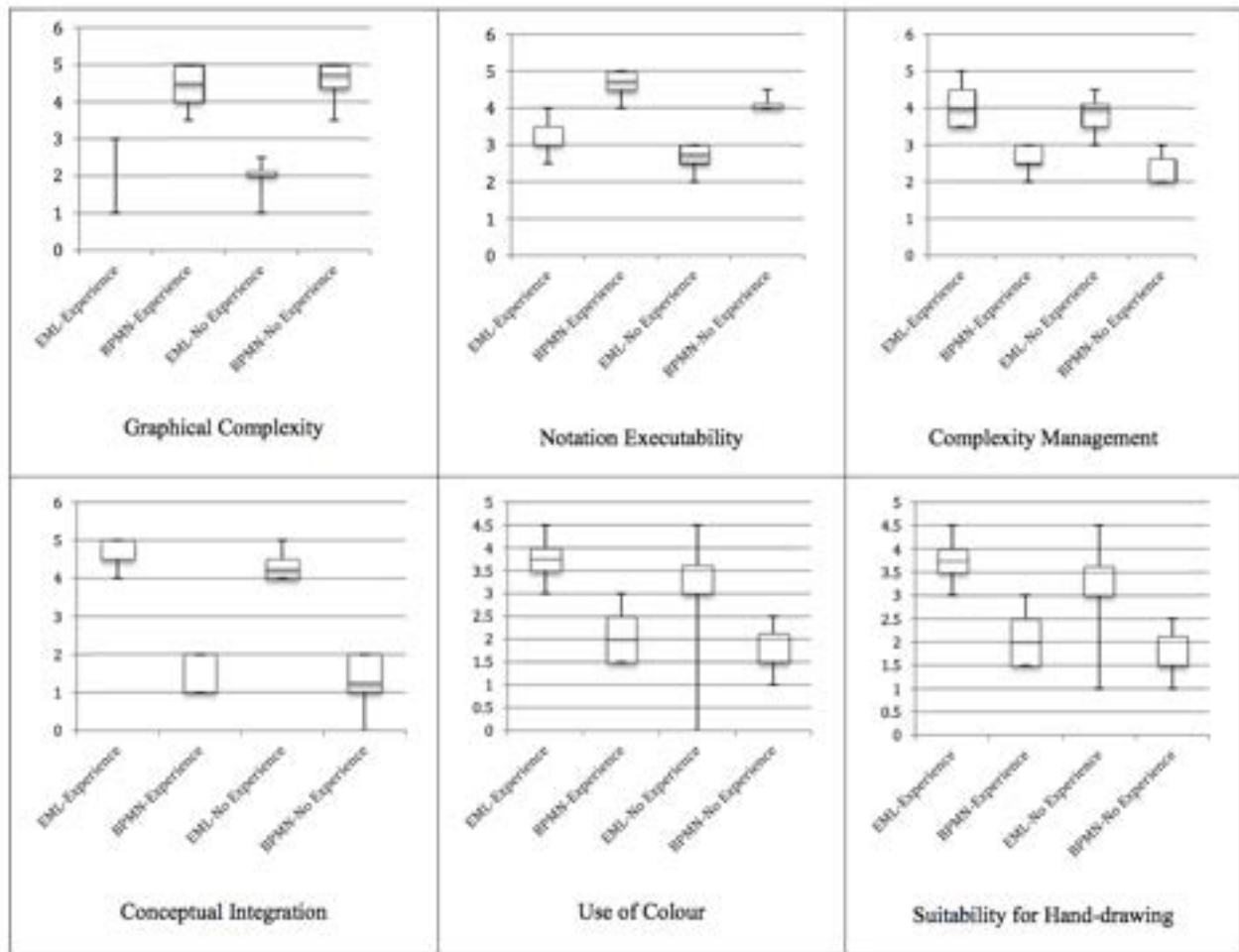


Figure 16: User Perception of EML Vs. BPMN using several dimensions.

Graphical Complexity

For graphical complexity, the responses range from 0, the notation is not graphically complex, to 5 the notation is extremely graphically complex, the one feature where smaller is better. A majority of both BPM experienced users and non-experienced BPM found the graphic complexity of BPMN (both groups mean 4.5, std. devn. 0.5) to be extremely high and thus to be a barrier to its learning and use. By contrast, with EML (experienced mean 1.9, std. devn. 0.4; no experience mean 2.0, std devn. 0.5), they found it much easier to learn to use each symbol, use it correctly and combine it with other EML symbols. There was no significant difference in the responses of those with BPM experience and those without. The BPMN notation library has a total of 177 visual symbols. Such a level of graphic complexity is clearly problematic for users. EML only has 15 symbols, making the notation easier to learn, apply and understand. In BPMN, the number of grammatical rules increases with the number of symbols. The specification contains hundreds of rules for how symbols can be combined together. However, EML only has 10 rules in total.

Notation Executability

BPMN (BPM experienced mean 4.7, std. devn. 0.3; no experience 4.1, std devn. 0.2) rated significantly better for notation executability in comparison to EML (experienced mean 3.2, std. devn. 0.5; no experience mean 2.6, std devn. 0.4). BPMN's execution semantics means that process engines can execute BPMN visual models, a strong model-driven development capability that our subjects found appealing. Although EML can be used to generate BPEL code in MaramaEML, and MaramaEML has an integrated LTSA engine to validate the

generated BPEL quality, we have not achieved the same level of formal mapping and standardization as is the case with BPMN. However, this was never our intention. We are not trying to produce a rival notation for BPMN or repeat the things that BPMN is doing well, but to use EML as a complementary notation to improve the usability and enhance the modelling ability. Those without BPM experience rated both BPMN and EML's notation executability significantly (at the 0.05 significance level) lower (by 0.4-0.5 scale points on average) than did those with BPM experience. This perhaps indicates a lack of knowledge as to what types of execution of a business modelling language might be useful.

Diagram Complexity Management

EML (experienced mean 4.1, std. devn. 0.5; no experience mean 3.8, std devn. 0.5) scores significantly better than BPMN (experienced mean 2.6, std. devn. 0.4; no experience 2.2, std devn. 0.4) for complexity management. There was no significance in this rating between those with and without BPM experience. When using BPMN most subjects used "Page Connectors" and "Sub-elements" (e.g. Subprocesses, Subchoreographies and Subconversations etc.) to manage diagram complexity. Page connectors allow diagrams to be linked across multiple pages, and "Sub-elements" allow users to break down the diagram into smaller elements. However, even expert users claimed they were not aware of other approaches in BPMN to reduce diagrammatic complexity, in spite of the availability of BPMN sub process diagrams. One user claimed the Page Connector was not an effective solution as it requires users to move between pages, potentially involving a loss of context in the transition. As for "sub-elements", while users generally believed that there is a good advantage in using our diagram nesting approach, most of them do not think a single solution (one user calls it an "all in one" solution) will be enough to resolve the complexity issue in BPMN.

EML uses a hierarchical decomposition approach to manage diagrammatic complexity. EML provides "sub-elements" via "Collapse" or "Expand" tree branches, and "page connectors" via the connection between different process overlays or tree branches. EML's unique overlay structure (process, trigger and exception) gives users the power to either expand all the details in the same diagram (if required), selectively expand, or expand them to separate diagrams or processes at the next level of detail. EML users have a set of hierarchically linked diagrams rather than a single diagram with smaller diagrams nested inside. It is unclear to us why subjects chose not to use BPMN sub-process diagrams in a similar manner. Most subjects reported that although EML potentially increases the number of diagrams compared to BPMN, all of them are cognitively manageable in size. More than 90% of the participants expressed that they preferred this approach instead of a "super large" diagram as very often happens with BPMN and similar graph-based notations. Other empirical studies (Moody 2002) show that decomposing big diagrams into linked hierarchical diagrams can improve end user comprehension and verification performance by more than 50%.

Conceptual Integration

Our evaluation results show that EML (experienced mean 4.6 std. devn. 0.3; no experience mean 4.3, std devn. 0.4) provides a significantly better conceptual integration capability than BPMN (experienced mean 1.6 std. devn. 0.5; no experience mean 1.3, std devn. 0.8). Subjects found that BPMN diagrams are limited in scope to a single end-to-end business process, resulting in a large number of diagrams at the same level of abstraction with no easy way of gaining an overview of the domain as a whole. When we asked subjects to provide an overview model for the example (in step 5 of the evaluation), they either left it empty or developed their own informal diagrams together with BPMN diagrams to deliver the overview. For EML, the participants reported that the integration of tree and overlay structure provides a very good way to represent the big picture, from the concept level overview to the detailed activities.

Colour-based Visual Expression

EML had a significantly higher result (experienced mean 3.75 std. devn. 0.4; no experience mean 3.1, std devn. 1.1) than BPMN (experienced mean 2.2 std. devn. 0.6; no experience mean 1.8, std devn. 0.5) for this feature. Those with BPM experience rated EML significantly (at the 0.05 level) higher for this feature than those without BPM experience, but there was no significant difference in the rating of BPMN. BPMN does not prohibit the use of colour, but just not actively encourage the user to use it. So because in BPMN every user can freely choose preferred colours, these vary from user to user, potentially leading to misinterpretations. Most of our subjects considered that the colour usage in their BPMN diagram was akin to artwork instead of being used to achieve cognitive effectiveness. More than 80% of our users expressed that EML was the first process modelling language that formally asked them to use the power of colour. Nearly 95% of subjects supported the idea of defining and using colour usage rules formally, as it helps them to enhance communication via the

modelling language. However, we also found one colour-blind user who had difficulty with the colour usage in EML.

Suitability for Hand-drawing

Our evaluation results showed EML (experienced mean 3.8 std. devn. 0.4; no experience mean 3.2, std devn. 0.9) to be significantly better than BPMN (experienced mean 2.2 std. devn. 0.6; no experience mean 1.8, std devn. 0.5) for this feature. Subjects found that the BPMN symbols set presents considerable challenges for hand drawing, as it includes a wide range of icons, composite symbols, shape fills and border styles. Subjects also found BPMN's in-situ decompositions difficult to apply, as any expanding of compound elements essentially requires redrawing the diagram. In contrast, most of the participants found that EML diagrams can be drawn quickly and easily. The multi-overlay structure allows users to add extra overlay on top of existing tree or processes instead of redrawing. However, expanding a tree branch requires some degree of redrawing overhead. For most EML diagrams, subjects only needed to remember 4 major symbols (service, task, tree structure and overlay), and use four different colour pens.

Overall, our comparative usability results provide significant support for hypothesis H2, ie. *Users find EML more usable than BPMN*, in all areas except notation executability. The results suggest that MaramaEML is straightforward to use and understand in comparison to BPMN.

6.4. Limitations

Several threats to validity of our study can be identified. Selection bias could result from the subject recruitment approach. We attempted to mitigate this by using a broad variety of mailing lists for our invitation and taking a first come first served approach to respondents. While the latter eliminates some forms of bias it could provide a bias towards those who have strong interests in BPM; this may explain the larger numbers of those with BPM experience compared to those without (20 versus 10 participants). In addition, it has created a clear bias towards those with formal IT experience over those with no formal IT knowledge (25 versus 5 participants). We have no way of knowing whether the proportions observed in our sample are reflective of the population at large using BPM routinely. Also, the number of participants with no formal IT training was too small for us to be able to differentiate in any significant way the responses of those with and without formal IT training. Repeating the experiment with a larger proportion of non IT trained participants would allow us to draw better distinctions between these two populations.

We chose to compare EML against BPMN only as an example of an existing box and connector BPM approach, suggesting an internal threat to validity over how representative BPMN is in this regard. The widespread use of BPMN suggests this threat is minimal. We also were unable to measure task durations. This was because we were trying to eliminate the external threat of experimenter effects by running the task sessions unsupervised. However, this means our measurement of performance in evaluating H1 only relates to quality of models and not task durations. For the latter we had to rely on qualitative feedback from subjects.

6.5. Research Questions

Returning to our two research questions, the evaluation suggests support for an affirmative answer to RQ1 (*can a tree-and-overlay based service modeling approach provide advantages over existing box-and-line based visual business process modeling languages?*). However, the strength of this conclusion is limited by the bias of the sample population towards those with BPM and IT experience. The usability feature comparisons demonstrate a clear usability advantage for EML over the use of BPMN, even for this sample group. For RQ2 (*can an effective and efficient environment be implemented for modeling, checking and generating service orchestrations from these tree/overlay-based models?*), the answer is a little less clear cut. MaramaEML does not have as strong a support for code generation as do existing BPMN tools and this was reflected in the Scalability and Efficiency scoring in the isolated usability results and also the Notation Executability results in the comparative study. Combined these indicate only a partially affirmative answer to RQ2, indicating further work needs to be undertaken to enhance efficiency and scalability of the toolset. Given the bias of the sample population to BPM experience, including experience with other BPM-supporting tools, this may have resulted in more negative answers overall than if a larger sample of non-BPM experienced users were sampled. However, we were unable to demonstrate an statistically significant difference between the groups from our evaluation results.

6.6. Future Enhancements

Based on our evaluation, several improvements and extensions could be made to the EML and MaramaEML to increase their flexibility and functionality. During our end user evaluation, we received strong demand for integrating UML views e.g. sequence diagrams, activity diagrams and state diagrams, into the tool, to complement the other notational views. This would require notation exchange between EML, BPMN and UML meta-models. The current version of MaramaEML includes basic consistency checking between EML views and BPMN views. We use an event log to keep the usage records and trace the model changes. If the user changes the name of a model element in an EML view, an event handler checks the corresponding mapped element in the BPMN view and updates the name automatically. However, this approach cannot cope with situations requiring complex logical analysis. For example, if the user tries to delete a node that has several sub-nodes in EML, the system at this stage will automatically delete all the sub-nodes in an EML view and all the corresponding nodes in the BPMN view. This solution is insufficient. To address this, we need to incorporate a more comprehensive consistency mechanism (e.g. link the sub-nodes with other nodes if there is an underlying relationship between them). We also intend to make use of the OCL-based technique provided in the Marama meta-tools (Li et al 2008) to define dependency and consistency constraints in the EML meta-model.

We are working on approaches to integrate MaramaEML's high level business process modelling with other tools we have developed in the software architecture and application level domains. MaramaMTE (Middleware Testing Environment) is a tool for modelling complex software architectures and generating performance test beds from these models to assess likely software architecture performance (Cai and Grundy et al 2007). We are exploring the possibility to integrate software architecture specification and performance modelling with MaramaMTE via the Marama platform. This would allow performance engineering of complex business process orchestrated web services. ViTABaL-WS (Li et al 2004) is a tool providing event-based web services composition, supporting modelling of both event-dependency and dataflow in designing complex web service compositions. Integration of EML-based business process modelling and ViTABaL-WS-based web service composition and co-ordination modelling would provide more powerful, comprehensive support for both process modelling and web service composition.

During our end user evaluation, we observed that an achromatopsia (colour blind) participant became totally lost in the overlay integration view. An overlay integration view normally mixes process flows (blue color), trigger flows (red color) and exception flows (green color) in the same view. With this version of EML, color is a very important design component to represent and distinguish the information. However, from the evaluation, we found this important piece of visual information disappeared when the notation was viewed by a color blind user. Future research is required to adapt the notation for this group of users probably by using some form of dual coding mechanism.

During the end user evaluation, we found that some users, with low performance computers had response issues with the tool when they were using the zooming or fisheye view function for large EML trees. Improved zooming and fisheye view algorithms are required to improve performance.

7. Summary

Visual languages play a critical role in the business process modelling field. They are used in all areas and all levels of business process modelling practice, from strategic planning to active design. BPMN has emerged as an international standard for business process modelling. Like most other business process modelling notations, BPMN uses workflow based visual approaches. Research shows that "cobweb" and "labyrinth" (Recker and Rosemann et al 2007) or "map shock" (Blankenship and D.F. 2000) problems appear heavily in the complex work flow based diagrams, and lead to cognitive overload. Meanwhile, our earlier work (Anderson and Apperley 1990; Phillips 1995; Li and Phillips et al 2004) on complex user interfaces and their behaviour modelling demonstrated that a tree-based overlay structure can effectively mitigate complexity problems. Other researchers also recommend hierarchical structure as the most cognitively effective way of managing the complexity of diagrams (Sliver 2000; De Marco 1978).

To the best of our knowledge, EML is the first tree overlay structure-based hierarchical decomposition visual language in the area of business process modelling. EML is a novel business process modelling language based on using a tree hierarchy to represent organisationally structured-services and overlay metaphors to represent process flows, conditions, iteration and exceptions. Complex business systems are represented as service trees. Business processes are modelled as process overlay sequences on these service trees. By combining these two mechanisms EML gives the user a clear overview of an entire enterprise system with business processes modelled by overlays on the same view. An integrated support tool for EML has been developed using the Eclipse based Marama framework. It integrates EML with existing business notations (e.g. BPMN) to provide high-level business service modelling. Functions for distortion-based fisheye and zooming

views enhance complex diagram navigation ability. MaramaEML can also generate BPEL code automatically from the graphical representations and map it to a LTSA-based model checker for verification.

Although we have used BPMN as a benchmark during the development of EML, we are not proposing EML as a replacement or rival notation for BPMN. In fact, we believe BPMN is an important and successful visual notation in the field of business process modelling. Our end user evaluation also shows BPMN and EML both have advantages and weaknesses, and most importantly, these advantages and weaknesses between BPMN and EML, or at wider level, between work flow based notations and tree overlay based hierarchical decomposition notations, are not overlapping, so there is a space for them to be complementary. Based on this constructive spirit, we developed EML to promote this complementarity and raise the possibility of hierarchical based business process notation.

Acknowledgements

The authors gratefully acknowledge the Foundation for Research, Science and Technology's support for this research under the Research For Industry fund, the University of Auckland for support of this research and Richard Li's PhD, and Jun Huh for his considerable work on Marama meta-tools making MaramaEML possible.

References

1. Adams, M., ter Hofstede, A.H.M., La Rosa, M.: Open Source Software for Workflow Management: The Case of YAWL. *IEEE Software* 28(3), Mar 2011, pp. 16-19.
2. Aguilar-Savén, R.S. Business process modeling – review and framework, *International Journal of Production Economics*, vol. 90, no. 2, 28 July 2004, Pages 129-149
3. Ali, N., Hosking, J.G., Huh, J. and Grundy, J.C. Template-based Critic Authoring for Domain-Specific Visual Language Tools, In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, Cornwallis, Oregon, USA, Sept 20-24 2009, IEEE CS Press
4. Anderson P.S. and Apperley M.D. (1990): An interface prototyping system based on Lean Cuisine, *Interacting with Computers*, vol 2, No 2, 217~226.
5. Baeyens, T. (2007, May 02) The state of workflow, <http://www.jbpm.org/state.of.workflow.html>
6. Baker, J. (2002, June 07) Business Process Modelling Language: Automating Business Relationships, *Business Integration Journal*, www.bijonline.com
7. Blankenship, J. and D.F. Dansereau (2000) The Effect of Animated Node-Link Displays on Information Recall. *The Journal of Experimental Education*, 2000. 68(4): p. 293~308
8. Box, D., et al. (2006, January 24) Web Services Eventing (WS-Eventing), <http://www.w3.org/Submission/WS-Eventing/>
9. BPMI (2010, March 18) <http://www.ebpml.org/bpml.htm>
10. Buchmann, A., Bornhövd, C., Cilia, M., Fiege, L., Gärtner, F., Liebig, C., Meixner, M., and Mühl, G. (2004) DREAM: Distributed Reliable Event-based Application Management, In *Web Dynamics*, Springer
11. Cai, R., Grundy, J.C. and Hosking, J.G. (2007) Synthesizing Client Load Models for Performance Engineering via Web Crawling, *IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, USA, Nov 5-9 2007, IEEE CS Press
12. Chen, P. (2002) Entity-relationship modeling, *Contributions to SE*, p296 ~ p310, Springer-Verlag, NY.
13. Citrin, W., *Strategic Directions in Visual Languages Research*. ACM Computing Surveys, 1996. 24(4).
14. De Marco, T. (1978): *Structured Analysis and System Specification*. 1978, Yourdon Press
15. Draheim D and G. Weber (2005) *Form-Oriented Analysis*, Springer-Verlag, Berlin Heidelberg.
16. Engels, G. and Erwig M. (2005) ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications, In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, Long Beach, CA, USA, pp. 124-133
17. Eriksson, H.E. and Penker, M., (2000). *Business modeling with UML: business patterns at work*, Wiley
18. Feng Yi-Heng and Lee C. Joseph (2010) Exploring Development of Service-Oriented Architecture for Next Generation Emergency Management System. *AINA Workshops*, 557-561
19. Foster, H., Uchitel, S., Magee, J. and Kramer, J. (2003) Model-based verification of web service compositions, In *Proceedings of the 18th IEEE international conference on automated software engineering*, Montreal, Canada, Oct 2003, IEEE CS Press.
20. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley
21. Genon, N., Heymans, P., & Amyot, D. Analysing the cognitive effectiveness of the BPMN 2.0 visual notation. In *Software Language Engineering*, Springer 2011, pp. 377-396.

22. Gillain, J., Faulkner, S., Jureta, I. J., & Snoeck, M. Using goals and customizable services to improve adaptability of process-based service compositions. In *IEEE Seventh International Conference on Research Challenges in Information Science (RCIS 2013)*, IEEE, May 2013, pp. 1-9.
23. Goel A (2006) Enterprise Integration --- EAI vs. SOA vs. ESB, Infosys Technologies White Paper
24. Gottfried, H. J. and Burnett M. M. (1997) Programming Complex Objects in Spreadsheets: an Empirical Study Comparing Textual Formula Entry with Direct Manipulation and Gestures, Seventh Workshop on Empirical Studies of Programmers (ESP'97).
25. Green, T. R. G. and Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, vol. 7, 131-174
26. Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *Information and Software Technology*, Vol. 42, No. 2, January 2000, pp. 117-128.
27. Grundy, J.C., Hosking, J.G., Li, L. and Liu, N. (2006) Performance engineering of service compositions, ICSE 2006 Workshop on Service-oriented Software Engineering, Shanghai, China.
28. Grundy, J.C., Hosking, J.G., Li, N. and Huh J (2008): Marama: an Eclipse meta-toolset for generating multi-view environments, Formal demonstration at the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008, ACM Press.
29. Grundy, J.C., Hosking, J.G., Li, N., Li, L., Ali, N.M., Huh, J. (2013): Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications, to appear in *IEEE Transactions on Software Engineering*.
30. Hanna, K. (2002) Interactive visual functional programmings, Proceedings of the seventh ACM SIGPLAN international conference on Functional programming ICFP '02, vol. 37, no. 9.
31. Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F. and Wilner, W. (1994) The Rendezvous Architecture and Language for Constructing Multiuser Applications. *ACM Transactions on Computer-Human Interaction (TOCHI)*. Vol. 1, no. 2, pp.81-125, ACM Press
32. Hudak, P. (1989) Conception, evolution, and application of functional programming languages, *ACM Computing Surveys*, vol. 21, no. 3, pp. 359-411
33. Jung, M.C. and Cho, S.B. (2005). A novel method based on behavior network for Web service composition, In Proceedings of the International Conference on Next Generation Web Services Practices, 22-26 Aug. 2005
34. Kim, J. , J.Hahn and H.Hahn (2000) How Do We Understand a System with (So) Many Diagrams? Cognitive Integration Processes in Diagrammatic Reasoning. *Information System Research*, 2000.11(3): p.284-303
35. Kornkamol J., S. Tetsuya, and T. Takehiro, (2003) "A Visual Approach to Development of Web Services Providers/Requestors", Proc. of VL/HCC'03, Auckland, p251~p253
36. Kraemer, F. A. and Herrmann P. (2007) Transforming Collaborative Service Specifications into Efficiently Executable State Machines, In Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)
37. Leymann 2001
38. Li, G., Muthusamy, V., & Jacobsen, H. A. A distributed service-oriented architecture for business process execution. *ACM Transactions on the Web (TWEB)*, 4(1), 2, 2010.
39. Li, L., Phillips, C.H.E. and Scogings C.J., (2004) Automatic Generation of a Graphical Dialogue Model from Delphi, Proc of APCHI 2004, Rotorua, p221~p230
40. Liu, N., Grundy, J.C. and Hosking, J.G. Integrating a Zoomable User Interfaces Concept into a Visual Language Meta-tool Environment, In Proceedings of the 2004 International Conference on Visual Languages and Human-Centric Computing, Rome, Italy, 25-29 September 2004, IEEE CS Press, pp. 38-40
41. Li L., Hosking J.G., and Grundy J.C (2007): Visual Modelling of Complex Business Processes with Trees, Overlays and Distortion-Based Displays, In Proceedings of the 2007 IEEE Conference on Visual Languages/Human-Centric Computing (VL HCC 07), Coeur d'Alène, Idaho, U.S.A
42. Li, L., Hosking, J.G. and Grundy, J.C. (2008): MaramaEML: An Integrated Multi-View Business Process Modelling Environment with Tree-Overlays, Zoomable Interfaces and Code Generation Demo session, In Proceedings of the 2008 IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 15-19 September 2008, IEEE CS Press.
43. Li, L. An Integrated Visual Approach for Business Process Modelling, PhD Thesis, University of Auckland, 2010.
44. Marshall C. (2004) Enterprise Modelling with UML. Designing Successful Software Through Business Analysis, Addison Wesley
45. Martinez A and M. Patino (2005) "ZenFlow: A Visual Web Service Composition Tool for BPEL4WS", Proc of VL/HCC'05, Dallas, P181~P188

46. Moody, D.L. Complexity Effects On End User Understanding of Data Models: An Experimental Comparison of Large Data Model Representation Methods. in Proceedings of the 10th European Conference on Information Systems (ECIS' 2002.) Gdansk, Poland
47. Moody, D.L., What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. in Proceedings of the 15th International Conference on Information Systems Development (ISD 2006). 2006. Budapest, Hungary.
48. Moody, D.L., The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 2009. 35(5): p.756-777
49. Nordbotten, J.C. and M.E. Crosby, The Effect of Graphic Style on Data Model Interpretation. *Information Systems Journal*, 1999. 9(2): p. 139~156
50. Pautasso, C. and Alonso, G. (2005) The JOpera Visual Composition Language JVLC, vol. 16, no. 1-2, pp. 119-152.
51. Phillips C.H.E. (1994a): Review of Graphical Notations for Specifying Direct Manipulation Interfaces. *Interacting with Computers*, 411~431
52. Phillips C.H.E (1995): Lean Cuisine+: An Executable Graphical Notation for Describing Direct Manipulation Interfaces. *Interacting with Computers*, 49~71
53. Recker, J., Rosemann, M., Indulska, M. and Green, P. (2009): Business Process Modeling: A Comparative Analysis. *Journal of the Association for Information Systems*, Vol. 10, No. 4, pp. 333-363
54. Recker, J. (2010): Opportunities and Constraints: The Current Struggle with BPMN. *Business Process Management Journal*, Vol. 16, No. 1, pp. 181-201
55. Recker, J. and Rosemann, M. (2010): The Measurement of Perceived Ontological Deficiencies of Conceptual Modeling Grammars. *Data & Knowledge Engineering*, Vol. 69, No. 5, pp. 516-532
56. Schnieders, A. and Puhlmann, F. (2005) Activity Diagram Inheritance. In Proc of the 8th ICBIS, Poland, p3 ~ p15
57. Silver, B., (2009) BPMN Method and Style: A Levels-based methodology for BPM process modeling and improvement using BPMN 2.0. Vol. New York. 2009: Cody-Cassid Press
58. Urbas. L., Nekarsova. L. and Leuchter. S. (2005) State chart visualization of the control flow within an ACT-R/PM user model, In Proc. IWTMD05, p43~p48.
59. Xiong, P., Fan, Y., & Zhou, M. A Petri net approach to analysis and composition of web services. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 40(2), 2010, 376-387.
60. Winn, W.D. (1993) An Account of How Readers Search for Information in Diagrams. *Contemporary Educational Psychology*, 1993. 18: p.162~185

7.5 A suite of visual languages for model-driven development of statistical surveys and services

Kim, C.H., Grundy, J.C., Hosking, J.G. A suite of visual languages for model-driven development of statistical surveys and services, *Journal of Visual Languages and Computing*, Elsevier, Vol 26, Feb 2015, pp 99–125

DOI: [10.1016/j.jvlc.2014.11.005](https://doi.org/10.1016/j.jvlc.2014.11.005)

Abstract: Objective: To provide statistician end users with a visual language environment for complex statistical survey design and implementation. Methods: We have developed, in conjunction with professional statisticians, the Statistical Design Language (SDL), an integrated suite of visual languages aimed at supporting the process of designing statistical surveys, and its support environment, SDLTool. SDL comprises five diagrammatic notations: survey diagrams, data diagrams, technique diagrams, task diagrams and process diagrams. SDLTool provides an integrated environment supporting design, coordination, execution, sharing and publication of complex statistical survey techniques as web services. SDLTool allows association of model components with survey artefacts, including data sets, metadata, and statistical package analysis scripts, with the ability to execute elements of the survey design model to implement survey analysis. Results: We describe three evaluations of SDL and SDLTool: use of the notation by expert statistician to design and execute surveys; useability evaluation of the environment; and assessment of several generated statistical analysis web services. Conclusion: We have shown the effectiveness of SDLTool for supporting statistical survey design and implementation. Practice implications: We have developed a more effective approach to supporting statisticians in their survey design work.

My contribution: Co-developed key ideas for this research, co-designed approach, co-supervised Masters student, wrote substantial parts of paper, co-lead investigator for funding for this project from FRST

A suite of visual languages for model-driven development of statistical surveys and services

Chul Hwee Kim
Fides Treasury Services Ltd.
Badenerstrasse 141
P.O. Box CH-8036
Zürich, Switzerland
ckim001@gmail.com
Phone +41-44-298-6559

John Grundy
Centre for Computing & Engineering
Software Systems
Swinburne University of Technology
PO Box 218, Melbourne, Australia
jgrundy@swin.edu.au
Ph: +61-3-9214-8731

John Hosking
College of Engineering and Computer
Science
Australian National University
Canberra, Australia
john.hosking@anu.edu.au
Ph: + 61-2-6125 8807

Abstract

Objective: to provide statistician end users with a visual language environment for complex statistical survey design and implementation. **Methods:** we have developed, in conjunction with professional statisticians, the Statistical Design Language (SDL), an integrated suite of visual languages aimed at supporting the process of designing statistical surveys, and its support environment, SDLTool. SDL comprises five diagrammatic notations: survey diagrams, data diagrams, technique diagrams, task diagrams and process diagrams. SDLTool provides an integrated environment supporting design, coordination, execution, sharing and publication of complex statistical survey techniques as web services. SDLTool allows association of model components with survey artefacts, including data sets, metadata, and statistical package analysis scripts, with the ability to execute elements of the survey design model to implement survey analysis. **Results:** we describe three evaluations of SDL and SDLTool: use of the notation by expert statistician to design and execute surveys; useability evaluation of the environment; and assessment of several generated statistical analysis web services. **Conclusion:** we have shown the effectiveness of SDLTool for supporting statistical survey design and implementation. **Practice Implications:** we have developed a more effective approach to supporting statisticians in their survey design work.

1. Introduction

Statistical surveys have extensive roots running back to ancient times [27]. The development of probability theory and mathematical statistics provided a scientific foundation for statistical surveys [1] and they have become a valuable and ubiquitous tool for obtaining trustworthy information. As such, statistical surveys are a very common tool for obtaining trustworthy information about a set of objects comprising a target population in many diverse domains, including market research, government policy development, and academic research. The goal of a survey is to describe the population by one or more parameters defined in terms of measurable properties. This in turn requires a *frame*, providing access to the population (eg a phone book or electoral roll), and a method for sampling from that frame [1]. Figure 1 illustrates the typical iterative process involved in defining and executing a statistical survey.

In addition to the survey process, other characteristics of a survey that need to be modelled include survey data, data analysis and relationships between process, data and analysis, and low-level data collection and analysis tasks. Many tools support aspects of realizing the survey process, including SurveyCraft [41] and Blaise [42] for questionnaire design and response collection, SAS [39] for complex data analysis, and SPSS [40] for general statistical process design. However most statistical survey support tools have been limited to supporting only parts of the surveying process [8][17], typically low-level statistical *technique implementations* including data capture, processing and analysis. Many other aspects such as statistical metadata, heterogeneity in statistical data semantics, and other non-mathematical activities are less well addressed [10], [19]. Available tools are also usually narrow in their focus and there is no agreed notation for defining a statistical survey overall.

We have been developing a higher-level statistical survey process support tool incorporating a range of domain-specific visual languages to enhance statistical survey design, implementation and reuse [20], [23]. To this end we developed the Survey Design Language (SDL) and its support tool, SDLTool, an integrated set of visual notations aimed at supporting survey design in a similar way that the Unified Modelling Language (UML) [36] supports software development. Our aim was to design a set of visual notations for statistical survey design that fill a similar role as the Unified Modelling Language (UML) [37] does in software design. Through these notations we aimed to:

- make statistical survey design more accessible.
- improve the effectiveness and efficiency of implementing statistical surveys
- provide better tool support for survey designers



Figure 1. Basic statistical survey process (from [2]).

Our research concerns practical issues in supporting survey processes. From a survey practitioner's viewpoint, statistical computing is well supported by high quality software packages like R [10], so low-level statistical technique implementation is not a major operational concern. However many other aspects such as statistical metadata [31] [19], heterogeneity in statistical data semantics [33], making complex techniques accessible and reusable by others, documenting complex survey processes and tasks, and other non-mathematical activities are less well addressed. SDLTool includes support for the generation of complex web services that provide data capture, processing and analysis support via off-the-shelf survey analysis tools. These survey implementations are generated as web services from high-level models and allow heterogeneous 3rd party applications to make use of these packaged techniques implemented by remote COTS statistical analysis tools. Generation of rich documentation for these packaged statistical survey services is also supported.

SDL evolved from an initial set of design notations to become a fully integrated environment that supports design, implementation and publication of the statistical survey process. Our hypothesis is that when the semantics of survey designs are visually specified and modelled in a visual language the following benefits are obtained:

- Mitigation of communication overheads for both non-experts and experts. A difficult survey concept can be represented with a graphical metaphor, an abstraction isolating low-level details from high-level concepts.
- Harmonisation of disparate operational semantics.
- Model-driven management and execution of the survey process.

We begin with a motivating example for this work, the New Zealand National Survey of Crime Victims, and from this review related work and define a set of key requirements we wanted to address. We then provide an overview of our approach and integrated environment and the SDL domain-specific visual languages. We use a usage example to explain the features of the SDLTool including the association of resources with SDL elements, the execution of fully defined statistical techniques and operations, visualisation of results, and publication of survey documentation and code. We then describe SDLTool's key architectural and implementation features. We present three evaluations we have carried out to judge how well this research has met our key requirements: an expert statistician usage and feedback on SDL; a user evaluation of SDLTool informed by Cognitive Dimensions theory [16]; and assessment of generated statistical analysis web services and documentation. We conclude with some key directions for future research in this area.

2. Motivation, Requirements and Research Question

The primary source of crime victimisation data for e.g. government agencies and the news media is usually law enforcement agencies. However the reality of victimisation may not be sufficiently captured from the one source and therefore victimisation data might usefully be supplemented by a statistical survey to:

- Better identify at risk groups.
- Provide an alternative measure of crime victimisation

- Describe both explicit and implicit effects of crime.

This broadly is the theme of the New Zealand National Survey of Crime Victims in 2001 [33]. The survey was a door-to-door survey targeted at New Zealanders aged 15 or more. Each region was assigned a unique area code, and a stratification of the area codes done. Random samples of strata were compiled and a pre-defined visit pattern was applied in collecting data e.g. 10th street, 10th house.

Figure 2 shows how the survey outlined above is depicted and elaborated using an SDL *survey diagram* (described in detail below). This was conceived as a brainstorming view for early conceptualisation of the survey design. Key *survey attributes* are clustered around a set of user-defined *survey contexts*. Together these visualise key processes in formulating the high-level nature of the survey. SDL provides a variety of such domain-specific visual language diagram types with which to model complex statistical surveys. These include survey diagrams (to model the overall context of a survey); survey task and data diagrams (to model data collection and analysis); survey technique diagrams (to model low-level details of statistical analysis processes); and survey process diagrams (to model high-level survey task interactions).

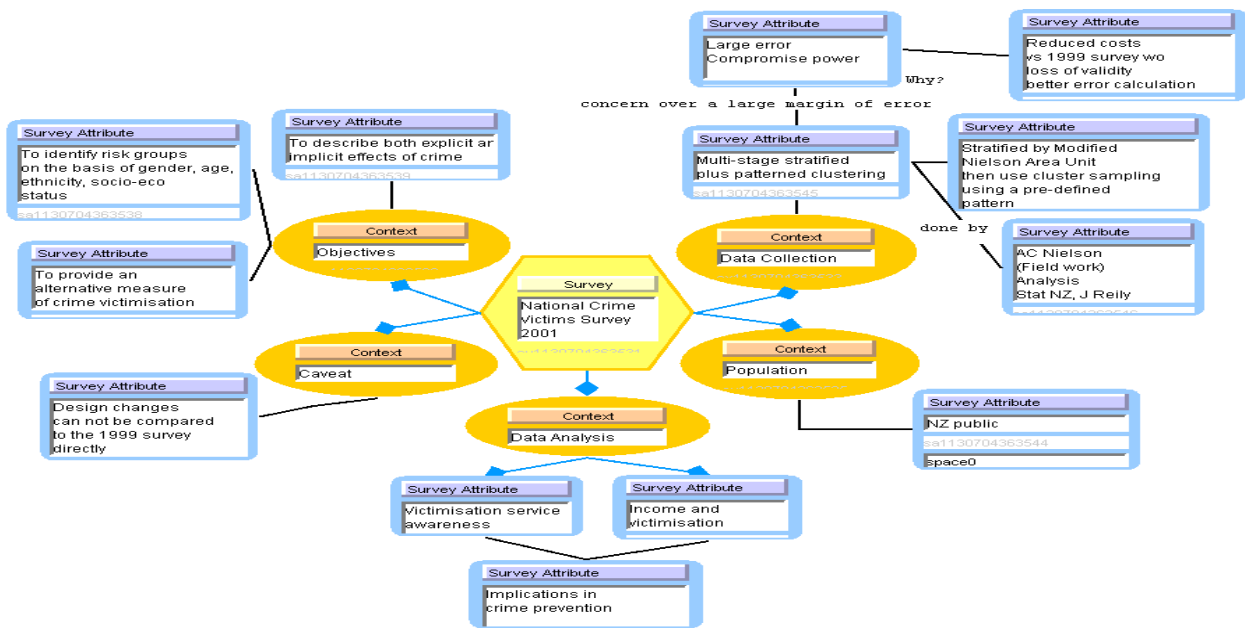


Figure 2. An example SDL Survey Diagram for the 2001 New Zealand Crime Victimization Survey

(© 2007 IEEE. Reprinted, with permission, from [22])

Currently developers of such statistical surveys have to build custom models of the survey process in an ad-hoc fashion outside their off-the-shelf analysis tools; build survey-specific data capture, processing and analysis *technique implementations* with multiple statistical analysis tools, such as R [17]; and often have great difficulty sharing these technique implementations as most analysis tools use proprietary representational techniques. Our background in the development of visual languages and environments suggested that an integrated visual language approach could address these problems and led us to formulating the following key research question, which we attempt to answer in the remainder of this paper:

RQ1: Can visual notations with appropriate software tool support be used by end users, i.e. statisticians and other survey developers, to facilitate statistical survey design and implementation?

3. Related Work

A variety of work has been done in the End User Development space using visual languages to support non-technical end users in developing complex solutions [26]. Common domains include but are not limited to supporting user-defined mash-ups, web and mobile application design and scripting, information visualisation, supporting end-user testing of apps and services, and web services composition. The most relevant to our work includes approaches to model domain-specific processes and tasks, and to script computational techniques. These include a variety of domain-specific languages for end users including business process modelling [24], biological modelling with state charts [20], ontology modelling languages [24], and visual composition of applications from basic building-blocks [10]. All of these approaches define one or more visual languages allowing target end users to describe designs and solutions in the target

domain using visual modelling abstractions. In our work, we want to enable end user design of complex statistical surveys including processes, tasks, data, and analytical techniques.

A range of approaches have been developed to support end user visual modelling of complex web service compositions. The most relevant to our work includes support for web service compositions using composition-oriented techniques [45], work on specifying service compositions for learning management systems [12], service-based user interface orchestration [11], and end user development of various government and business services [13]. These approaches aim to enable target end users to define and compose parts of complex service-based systems leveraging their domain-specific knowledge. These end-user defined or configured services can then often be reused by others. In our work, we want to enable packaging and reuse of survey techniques as services.

Statistical surveys have become an essential quantitative information gathering and analysis tool in a wide variety of domains. Statistical data is used in the development of products, the analysis of organisational and government performance, the understanding of audiences for TV and radio, political and economic commentary and decision making, and teaching and research within Universities. Designing and implementing a good statistical survey requires a range of skills and experience, and ideally good survey design support tools. Additionally, the inception of large-scale surveys, such as the US census, brought enormous managerial complexity prompting development of statistical computing [5]. However, designing and implementing good statistical surveys is challenging.

Current survey supporting tools can be generalised into three broad categories: Computing-centric tools, statistical applications, and interviewing and data collection aids. Computing centric tools, which are almost synonymous with statistical computing, build on domain specific languages such as S and execution engines. They provide an excellent numerical computing environment, but their steep learning curve and the high level of investment in low-level implementations required can offset their advantages. Statistical applications provide an environment where users can be insulated from the low-level complexity by introducing a user centred interface for setting up high-level activities. This approach helps users focus their efforts on carrying out the survey process itself, however the proprietary lock-in of operational procedures and back end functionality which overlaps with the computing centric tools, where statistical applications lack the power of dedicated computing tools, are notable trade-offs. Interviewing and data collection aids focus on the early stages of the survey process thus their benefits do not flow into other parts of the survey process. Thus, while existing survey tools are generally useful they all tend to address only specific parts of the survey process. This forces users to weave a set of tools that lack a semantics to describe the overall survey process.

A small number of tools aim to support the survey process more generally. ViSta [46] provides a visually guided and structured environment for data analysis with multiple visual models to suit various end user groups. GuideMaps help users carry out data analyses even when they lack detailed technical knowledge. WorkMaps visualise a data analysis capturing valuable contextual information such as analytical steps taken. Task based organisation is well expressed but there is little support for merging tasks to support survey contexts, such as objectives, and data collection and analysis or for non-analytic tasks e.g. questionnaire design and metadata support is lacking. Many software packages have been developed to support aspects of the survey process including SurveyCraft [41] and Blaise [42] for questionnaire design and response collection, SAS [39] for complex data analysis and SPSS [41] for statistical process design. However, such tools are typically narrow in their focus and the lack of integration and cross tool support, hindered by the absence of mature and widely accepted inter-operable standards, are common sources of difficulty in the survey process. ViSta's Lisp-based extensions are a barrier to use given the dominance of S-based languages. ViSta also has poor back end integration support.

CSPRO [8] assists users from the data entry stage to produce error and inconsistency free data. It supports data entry, batch edit, and tabulation applications with powerful logic programming support. It only covers some stages of the survey process, with no support for sampling design and data analysis. Thus it is not a solution platform for users to integrate multi-faceted aspects of statistical surveys but it is a very capable tool in the areas it covers.

Statistical data and metadata standard initiatives include Triple-S [19], DDI [9] and MetaNet [31]. These efforts tend to be localised or discipline-specific, but understanding them has given us useful insight into incorporation of statistical data and metadata into SDL.

Our work has been strongly influenced by UML [36], particularly its use of multiple notations, supporting multiple modelling spaces [44]. While the UML is too software domain-specific for direct use in survey design [31], the historical development of UML provides valuable insight into visual languages that deal with a complex multidimensional problems. It is no coincidence that the high level aims of SDL closely parallel that of UML: *Visualisation*: SDL notations and diagrams provide visual representations of a statistical survey, spanning the entire survey process, but with each diagram visualising a specific aspect of the survey; *Specification*: the SDL diagrams as a whole assemble specifications of survey artefacts, their attributes and relationships and resource descriptions mapping artefacts to physical resources and services; *Implementation and execution*: similar to UML's MDA approach, SDL diagrams can generate and orchestrate software solutions for a statistical survey; *Documentation*: SDL can largely automate the documentation of the survey process.

4. Our Approach

After initial informal discussion with statisticians, we postulated that a set of integrated, Domain-Specific Visual Languages (DSVLs), each designed to model specific aspects of statistical surveys, would provide statisticians with an appropriate set of high-level and low-level modelling facilities, matching their cognitive models of survey design. A support environment for the modelling of statistical surveys using these DSVLs was needed that must incorporate modelling of survey resources, associating resources to statistical survey design elements, running modelling surveys on the target population datasets, and providing a range of data visualisation support features. Generating reusable scripts for 3rd party packages eases the burden of producing such scripts, which are often quite complex and hard-to-maintain directly, and enables reuse of the modelled techniques. Generated web services providing remote statistical technique implementations then provide a set of analysis services, implemented by the existing 3rd party statistical analysis packages, for both applications and other analysis tools, without users needing knowledge of the implementation of the techniques. They also have the added advantage of allowing 3rd party analysis tools to be used by others without the need to have an installation of the tools themselves.

Figure 3 illustrates the process of designing, generating and using statistical technique implementations using SDL and its supporting SDLTool. One or more repositories store statistical survey meta-data and/or SDLTool survey models (1). Some of these repositories may be 3rd party while others might be SDLTool-specific. Users import one or more useful SDL model templates and/or meta-data formats from these repositories and then develop their own statistical survey designs (2). SDLTool supports collaboration with other users by sharing SDL models and diagrams using a CVS repository and synchronous collaborative editing (3). Once a statistician has completed detailed survey technique specification, scripts are generated by SDLTool (4) that drive external 3rd party statistical processing and analysis packages, such as R [17]. After testing these survey technique implementation scripts the user instructs SDLTool to generate web service interface implementations (currently in Java using the Java Web Service Development Kit framework and tools) (5). Additionally WSDL descriptions of these services are generated and deployed to a web host, currently Axis (6). External tools such as existing statistical survey packages or bespoke Java or .NET applications can discover and invoke the packaged survey technique implementations via their web services (7). Required data is passed to the scripts via the generated web service interface, and result data is similarly packaged by the generated web service implementation and returned to the 3rd party application.

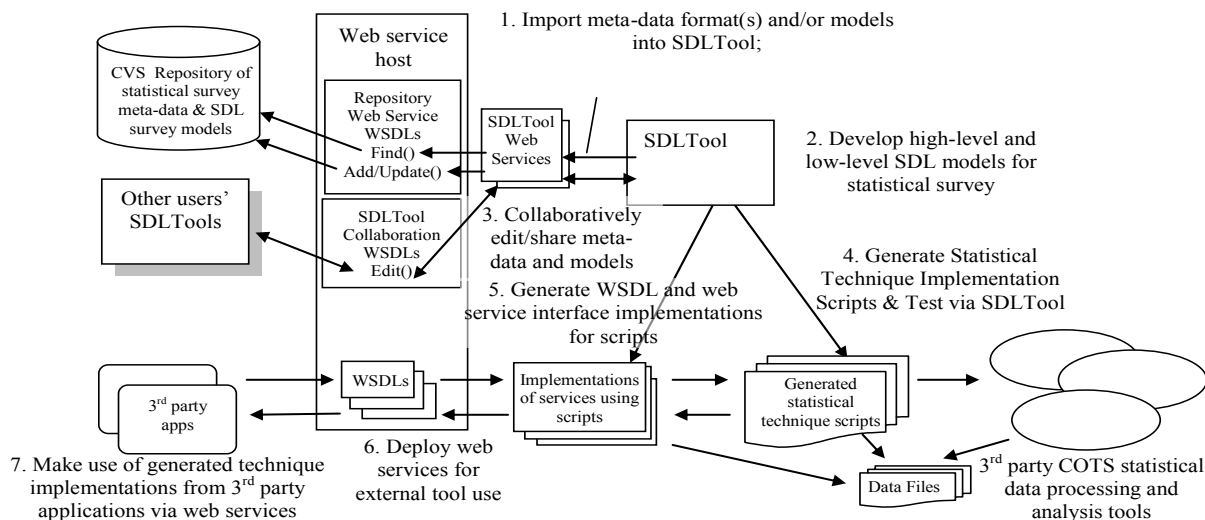


Figure 3. SDLTool usage overview.

Our approach to designing SDL was informed by several overlapping methodologies and our interest in answering the key research questions from Section 2. The first approach we used was Liu and Liebermann's of extracting semantics from natural language [28]. This was applied by examining a corpus of existing surveys and extracting key design elements and relationships. This helped establish an ontology for survey design and expressed key "building blocks" for statistical surveys. We used this ontology to provide the key elements for our underlying SDL meta-model and its associated visual languages. We incrementally enhanced this set of concepts using task analysis and from initial feedback showing statisticians our prototype SDL language designs.

The second approach we used was due to what we observed to be the close relationship between the requirements of survey designers and those of process centred software engineering environments [15], an earlier area of our research. Surveys are very process-centric, with each stage in a statistical process having associated resources (data), processing agents (human and machine), and so on. This suggested a process-oriented viewpoint would be an important component of SDL and would be used to link elements from data and analysis viewpoints. This led us to define SDL support for the process of survey construction based on earlier, successful software process support approaches.

A user-centric design approach was our third methodological tool [5]. We informally interviewed several experienced statisticians and asked them to walk through some example survey design tasks. These ranged from Professors of Statistics at the University of Auckland, to post-graduate statistics students, to non-statistician, more informal users of statistical surveys. We then conducted a task analysis to better understand what they did, when they did it and why they did it. From these identified tasks, we identified concepts in our ontological model that were being defined, refined and related during different statistical survey design tasks. Our task analysis clearly showed that survey construction often progresses “bottom-up” from a set of loosely connected goals and analysis tasks. During this task analysis we realised that our new SDL languages thus must strongly support “brainstorming” for determining survey objectives and survey design refinement. Consequences of this mean that users will incrementally define and build SDL diagrams in varying orders and completeness, and will iteratively refine and extend these models in practice. Additionally, we incrementally refined with these statisticians our initial findings from the ontological concepts and process-centric language examples above, and early SDL notation designs, described in the following section.

By application of these approaches, we identified, in conjunction with our statistician end-users, the following set of requirements needed to support more effective statistical survey technique definition, execution and sharing between analysis tools:

- *Modelling of statistical surveys.* Defining complex statistical surveys is very challenging with no agreed notations for many aspects of the survey e.g. overall process, particular techniques to use, specific tasks to carry out. As part of this modelling, informal brainstorming to determine survey objectives and survey design refinement is needed.
- *Association of Models with Resources.* An abstract survey definition needs to be instantiated with a myriad of specific population information sources, data collection and management tasks, statistical analysis technique implementations, coefficients, and result visualisation techniques.
- *Execution of surveys.* A support environment needs to provide exploratory task and technique application to partial datasets.
- *Generation of statistical technique implementations.* Once a set of statistical techniques has been fully specified, scripts implementing these for 3rd party statistical analysis tools can be generated. These scripts, often parameterised, can then be run to reuse the modelled techniques.
- *Wrapping of generated technique implementations in web services.* Tool users should be able to use techniques based on diagrammatic specifications outside of the environment. To make these technique implementations accessible to other tools, web services wrapping them need to be generated and deployed for discovery and invocation.
- *External tool interaction with generated technique web services.* 3rd party statistical tools can discover and invoke the generated statistical technique implementations via their generated web services.

Finally, we were strongly influenced by Burnett et al’s guidelines for robust visual language development [6], particularly their four ideal characteristics for visual languages: fewer concepts, explicit depiction of relationships, a concrete programming process and immediate visual feedback. This led us to use general principles of multiple, separate notations with small numbers of clearly differentiated symbols; multiple levels of abstraction where appropriate; and clearly defined semantics for each notational symbol for each SDL diagram type.

5. Statistical Design Language (SDL)

Based on our findings outlined above, we developed a design for SDL that supports five diagram types, one of which supports two levels of abstraction. Figure 4 shows the relationships between the five main SDL notations. These are:

- *Survey diagrams* - provide an overview of a survey and support collaborative brain-storming of the overall survey design. The central element represents the survey, with the key survey contexts (eg the survey objectives) and the key survey attributes (eg individual objectives) linked from each survey context;
- *Survey data diagrams* - describe at two levels of abstraction the structure of survey data and operations that construct and transform this data. They are used to model datasets, metadata and data operations (e.g. sampling). The metaphor is dataflow, with data sets manipulated by sampling operations to generate sampled data sets. Data diagrams may be “drilled into” for more detailed information about their data structures.
- *Survey technique diagrams* - define in more detail a statistical technique’s task sequencing /dependencies. These also use a dataflow metaphor, however, whereas survey data diagrams perform operations, such as sampling, which, in a statistician’s eye, modify the data, technique diagrams describe non-modifying data analysis probes such as regression analysis or multivariate graphing.
- *Survey task diagrams* - define specific tasks to carry out for data analysis and related activities. These hierarchically represent tasks undertaken in the survey. They may be instantiated from a template representing a reusable survey design and can be viewed as a specialisation of Sutcliffe’s task model [43]. Tasks may be associated with (bound to) survey artefacts such as data sets and reports.

- *Survey process diagrams* - define statistical procedures and the processes they are composed from and model the dynamic relationship between tasks. These aggregate tasks into stages and represent both flow between stages and use of task outputs as inputs to subsequent stages. Process diagrams can be “drilled into” to understand detailed relationships between a stage’s tasks.

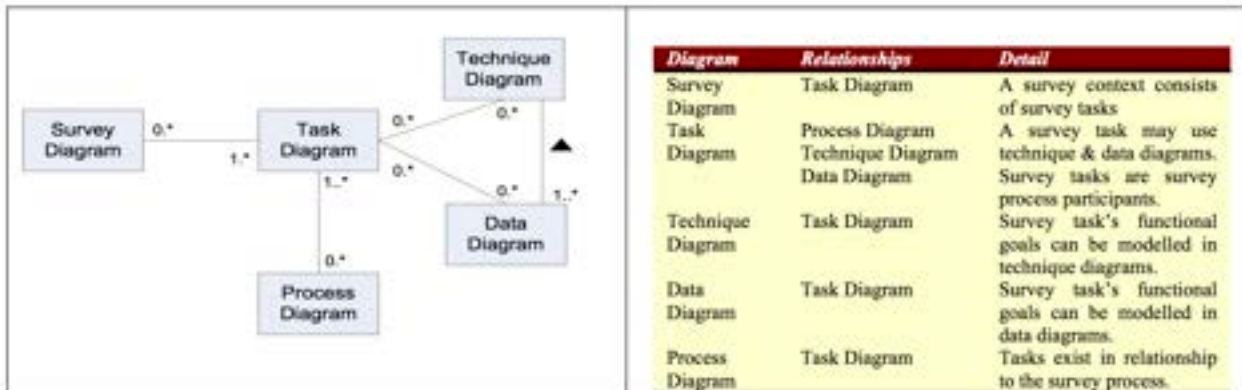


Figure 4: Relationships between SDL diagrams
(© 2007 IEEE. Reprinted, with permission, from [22])

There are six main design principles that shape the visual aspects of the SDL diagrams:

- Diagrams should be usable without software tooling. E.g. no elaborate layout schemes should be enforced.
- Terseness; in terms of a number of visual primitives
- Visual primitives in the context of a diagram are orthogonal and this results in having “one way of doing a specific task”.
- All default colour schemes can be overridden thus no colour has any symbolic value.
- A SDL diagram can be executed and all significant run-time changes during an execution of a SDL invoke visual changes that are either change in colour or border thickness.
- The SDL diagrams that map closely to statistical computing, Survey Technique and Data diagrams make use of a data-flow metaphor and they emphasize the functional model involving datasets and statistical functions that operate on them. This accounts for lack of visual primitives that map to imperative operations.

In the following we describe each diagram type in more detail using the 2001 New Zealand Crime Victimization Survey.

5.1 Survey diagrams

Survey diagrams provide an overview of a survey in terms of the various contexts it is organised by and the key survey attributes of those contexts. It supports interactive brainstorming to identify key aspects of a survey such as its requirements implications, analytical methodologies and time scale. Figure 2 (from Section 2) shows the survey diagram for the New Zealand Crime Victimization Survey. The survey diagram consists of an icon representing the survey (hexagon), contexts by which that survey is organised (ovals connected to the survey) and a hierarchy of survey attributes for each context (text boxes connected in a tree by arrow connectors. Survey diagrams attempt to lower an entry barrier in terms of accessibility. This is particularly important for a survey diagram as it is expected to be used by the most diverse set of survey participants in the context of a survey process. A survey diagram has five visual primitives and has no visual primitives that represent statistical techniques. There is no diagram type like this in other existing notations e.g. UML, BPML etc.

The Survey diagram notation is summarised in Figure 5. The survey diagram in Figure 2 shows us that:

- The objective of the survey is to find out (i) crime impact on victims; (ii) key risk groups; and (iii) alternative measure of victimisation [top left of diagram];
- The survey’s target population is the New Zealand public [bottom right];
- A caveat is that it can’t be compared to previous surveys due to changes in method [bottom left];
- To collect data, victims are stratified by Nielsen Area Units then clustered, and samples are selected randomly from each stratum [top right]; and
- Data analysis will attempt to determine if any relationships exist between income and awareness of victimisation service, and if so they will be tested using using discriminant analysis [bottom].

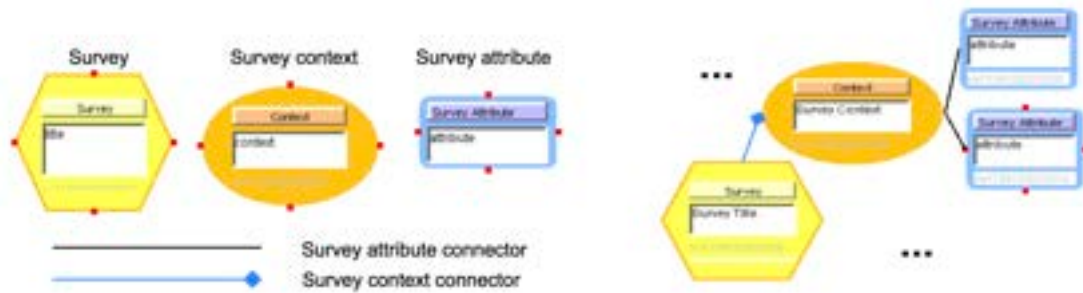


Figure 5. (Left) Survey diagram notation and (right) example usage.

A survey diagram’s scope covers the entirety of a survey but expressed at a very high-level. There are no rules as to how abstract or explicitly a context is expanded. The relaxed approach has both advantages and disadvantages due to the varying levels of abstraction across contexts. This is discussed later.

5.2 Survey task diagrams

Survey task diagrams are used to abstract a survey into as a set of tasks to be carried out to realise the survey. They mirror typical knowledge-based research activities. Both visual and meta-model aspects of a survey diagram are influenced by the generalised task model developed by Sutcliffe [43]. The main visual primitives are chosen so that the breaking down of a task into sub tasks can be elicited with a simple box and arrow diagramming technique. Our statistical survey task model is a composite of discrete (or other packaged composite) tasks. This model shows how these surveying tasks are chained together via relationships to form cohesive task models. The representation of task models in task diagrams takes the form of a collection of rectangles (the survey tasks) directed connectors (the sub task connectors) and parallelograms (the survey artefacts). The hierarchical order between tasks is explicitly annotated by the use of the task connectors. We considered using adaptations of BPML process diagrams or activity diagrams from the UML. BPMN diagrams have many additional features to support much more complex business modelling domains. Unlike activity diagrams, we wanted to mix flow and hierarchy on the same diagram type and also link survey artefacts to tasks in the visual model.

The default mode of processing task diagrams is from the left to right by convention. That is sibling tasks at the left side precede those at the right side sequentially. A task is considered fulfilled only if all sub tasks accomplished their goals. The expressiveness of the task execution order is extended by the use of two types of task operators: OR and Iteration. If the OR operator is present, an upper-level task may be carried out by one of several variations. The Iteration operator signifies that a task is on-going and that it can be done in parallel to other tasks.

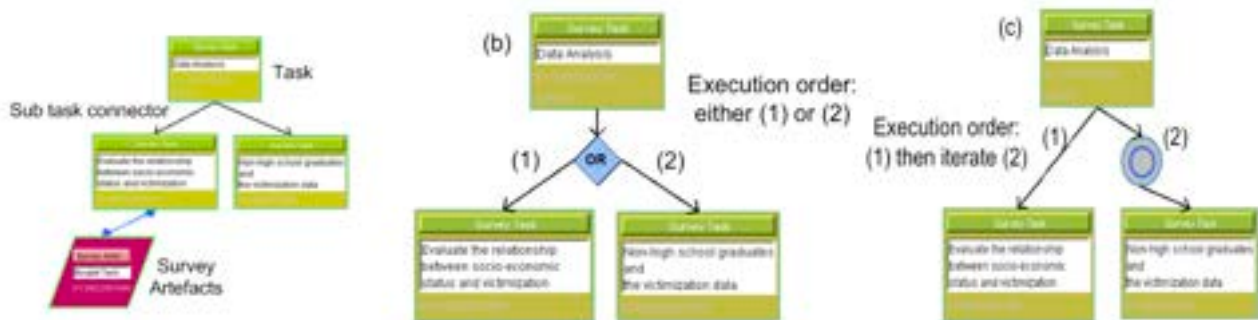


Figure 6. (a) Basic survey task diagram notation; (b) OR grouping; (c) iterative tasks.

Unlike a survey technique diagram, a survey task diagram does not have detailed specifications of its own and is not executable. For example, the iteration operator cannot be specified with a termination condition, which in consequence makes it infeasible to create an executable control flow out of it. However it is a very important diagram to co-ordinate together various aspects of statistical surveys expressed by the other three types of SDL diagrams: survey, survey data and survey technique. Survey tasks elaborate survey contexts (mapped from survey diagrams) and the survey artefacts they use map to survey datasets and survey techniques modelled by survey data diagrams and survey technique diagrams respectively. Therefore it provides an integration point for visually disparate types of SDL diagrams.

Figure 7 shows two survey task diagrams that show how two of the Crime survey contexts from Figure 2 (Data Collection and Data Analysis) map into tasks which are then decomposed into subtasks. The “Select Sampling Method” subtask highlighted in the Data Collection task decomposition is associated with a survey data diagram, as elaborated in the next section. In the Data Analysis task decomposition at right, relationships are also established to survey artefacts

corresponding to survey techniques to be used to carry out these subtasks. These survey techniques are associated with a survey technique diagram, shown in thumbnail form to the right, which elaborates how the techniques are composed together. This is explained in more detail in section 5.4.

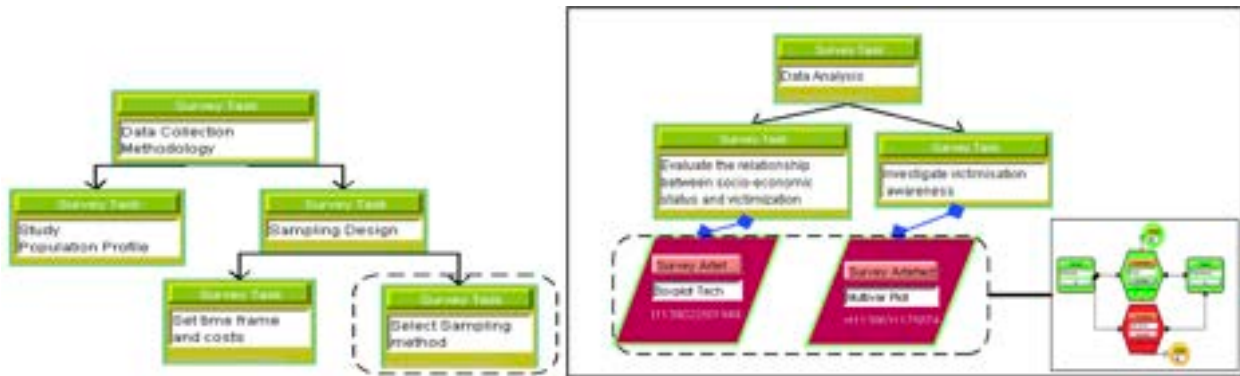


Figure 7. Basic crime survey task decompositions.

5.3 Survey data diagrams

Survey data diagrams provide a visual framework to support design of statistical sampling and data collection processes. They represent both the semi-structured data involved in the survey and data operations converting one data structure to another. The most important design aspect of the Survey Data diagram is its closeness of mapping to data collection activities from the viewpoint of a domain expert. All visual primitives of a survey data diagram map only to collected data, collection/sampling operations and output data that is filtered by such operations. Survey data diagrams can be executed and contain visual primitives and cues that are specifically designed for run-time outcomes. Survey data diagrams along with survey technique diagrams are the most immediately familiar form of SDL diagram to survey statisticians and data analysts alike, as they are familiar with data manipulation (specified in SDL data diagrams) and data processing and analysis (specified in SDL technique diagrams).

Survey data diagrams model statistical datasets, metadata and data operations (e.g. sampling). Major survey design concerns addressed by survey data diagrams include the visualisation of data flows, data operations, statistical metadata and the population to statistical data relationships. Survey data diagrams show data-level details as well as data-level operations that are usually derived from sampling techniques. In addition to showing statistical datasets using just one visual layer, survey data diagrams allow the relationship between statistical datasets and their metadata to be described using multiple diagram layers. Survey data diagrams are very different to ER model and UML class diagram data models in that they include both structural definitions and also include dataflow semantics. We chose to separate operations into standalone icons in a similar manner to other visual dataflow languages like LabView. We also chose to break out dataset structural elements as separate entities in response to user feedback that these are often critical to clearly see and work with. As survey data diagrams are translated into executable data processing services, they require sufficient dataset and operation properties to enable this service script generation.

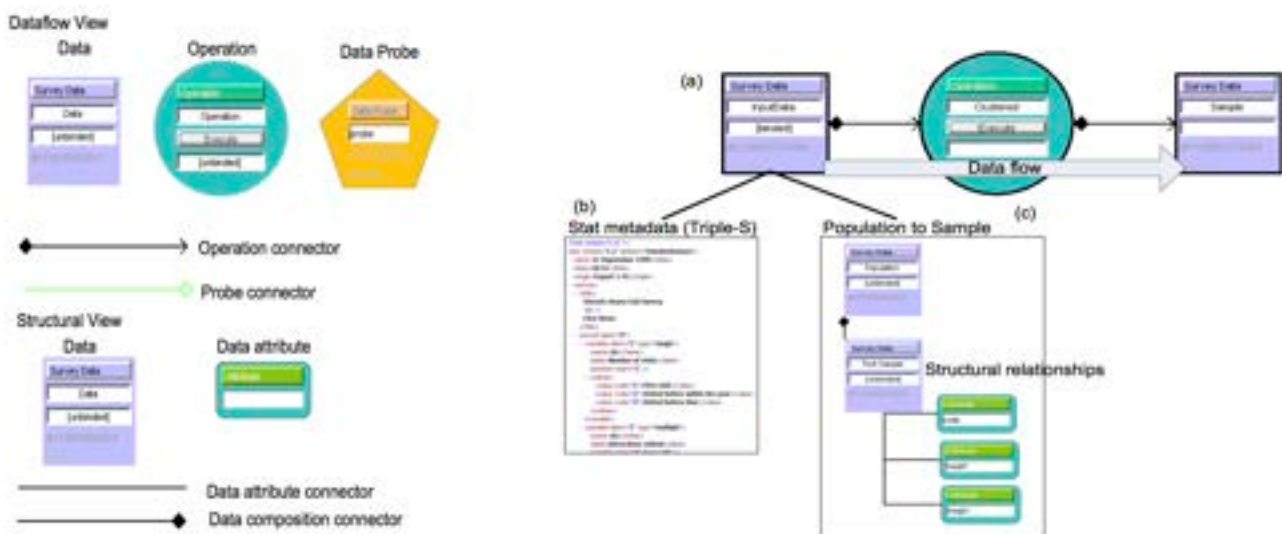


Figure 8. (Left) Survey data diagram notation and (right) example of usage in Crime survey.

Figure 8 (left) shows the main notational constructs in the Survey data diagram. The main theme of a survey data diagram is set by datasets (rectangles) and data operations (ovals). The pentagon icon represents data probing activities such as obtaining descriptive statistics on a dataset. Figure 8 (right) shows an example of the dataflow metaphor used and multi-layers of data encapsulated in data diagrams. (a) shows data described by a rectangle being processed by an operation to produce a new dataset. (b) is the underlying meta-data (Triple-S format in this example) being encapsulated. (c) shows the population we want to sample, with various features of the sample specified.

Figure 9 shows two examples of data diagrams from the crime victimisation survey design. The left side specifies the initial survey data source (Data Frame) and an initial 2-stage stratification operation (1) to produce the survey dataset (Data1). This is then further processed using a patterned clustering operation (2) to produce the final survey dataset (Sample). A mock statistical dataset can be bound to the ‘Data Frame’ icon for pre-testing purposes. In the post data collection stage, collected statistical data are bound to the ‘Sample’ icon. Shown on the right, stratified sampling produces a Sample. The next step in modelling the sampling process is to depict the sample to population relationships. This additional layer of information is associated with the survey data icon ‘Sample’. A range of data attributes are specified on Region and Subject.



Figure 9. Two Crime survey data diagram examples.

5.4 Survey technique diagrams

Survey technique diagrams specify an individual statistical surveying technique to be used in detail. A technique corresponds to a data processing or analysis technique usually embodied in a script run by a statistical analysis tool on specified data. Often these scripts can be parameterised for reuse. The survey technique diagram notation, shown in Figure 10, reuses the data flow metaphor and the entity tree notation of the survey data diagram. It adds information on the technique (logical entity), data consumed by it (specified by input ports), and the data produced by it (specified by output ports). Data is “bound” to these “ports”. The execution point (hexagon) to which the directed connector point represents the analytic technique and can be mapped to real-working version of the technique (e.g. an R procedure). All numerical dataset to dataset based flows are expressed with rounded rectangles and hexagon shapes and analytical results such as graphs are routed to oval shapes. This design choice visually annotates two types of data flows involved in using statistical techniques. One involves analytical outcomes of using a technique. Another involves piping of data from one technique to another.

While survey data and technique diagrams superficially appear similar, there are some significant differences in semantics and usage. In survey data diagrams the dataflow literally signifies the stream of datasets that are to be manipulated by sampling operations (e.g. selecting every nth data rows). This implies real physical changes in the datasets. However for survey technique diagrams no such analogy exists since generally statistical techniques imply only exploratory activities without explicit changes in datasets. Some examples of common statistical techniques are regression analysis, multivariate graphing, and multivariate data exploration. The survey technique diagram is another novel visual representation closer to visual functional languages such as Forms3 and Chameleon. However, we wanted to make the technique components clearly distinguishable i.e. data, technique, flow.

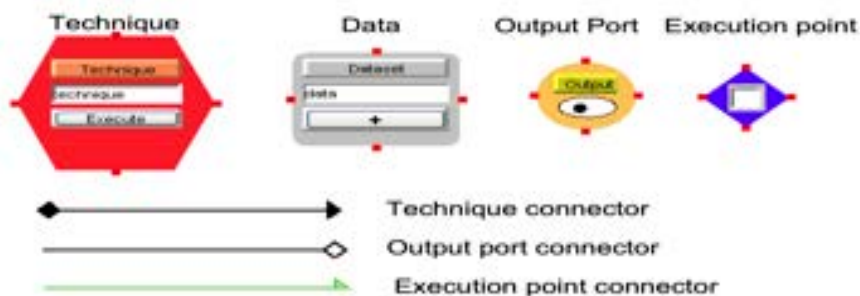


Figure 10. Statistical technique diagram notation.

Figure 11 shows two examples of the application of the survey technique diagram in the crime victimisation survey. Suppose we are interested in the level of education and crime victimisation in our crime survey example. To investigate the strength of association between two data variables, the use of the chi-squared approximation is chosen, as modelled via the left hand example SDL survey technique diagram. This has the sampled subjects as an input to the statistical technique ‘ChiSq’ and its output will decide the validity of the association. The ChiSq technique’s visual and textual outputs are directed to the output port and the data icon at the end of the dataflow is bound to the resulting data and metadata produced by the chi-square test.

Two of the goals in the data analysis stage for the crime victimisation survey are to: (1) investigate the relationship between socio economic status and victimisation; and (2) investigate the public’s awareness of victim support services and self-assessed safety. The collected data is published into a shared data repository and needs to be processed by two statistical techniques, boxplot and multivarplot, as shown in the survey task diagram Figure 7 (right) above. In this case study, the statistician wants to utilise visualisation methods to assess whether there is evidence of a statistical association between data variables. The two techniques are specified in the same diagram, shown in Figure 11 (right), as they consume the same statistical dataset.

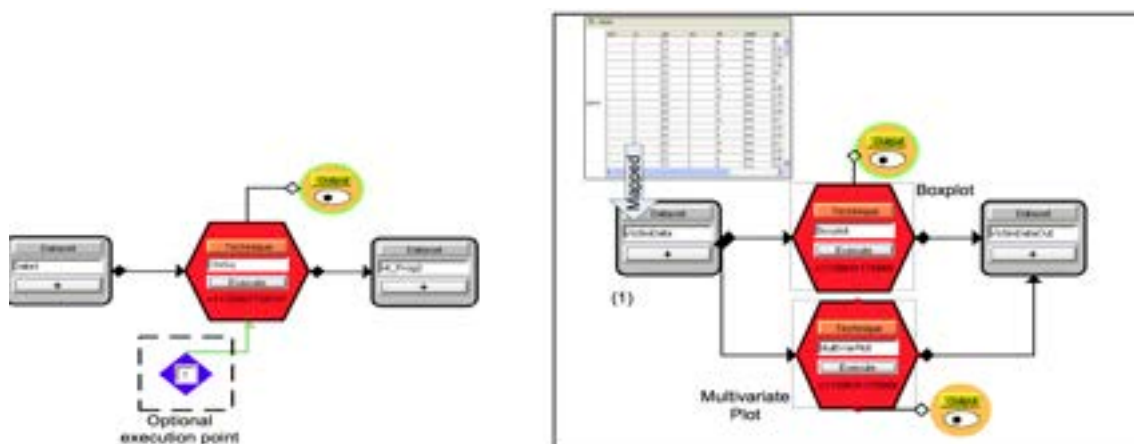


Figure 11. (left) Simple chi-square technique and (2) survey technique diagram for discriminant test.

5.5 Survey process diagrams

Having defined the high level overview of a survey and the data to be collected, the next step is to describe the analyses to be applied to the collected data using survey process diagrams.

A survey process diagram is used to denote various stages involved in a survey process and what tasks exist at each stage. Survey task diagrams manage each task independently but in survey process diagrams survey task to task relationship are expressed and their involvements in each stage is shown. Rectangular icons represent stages and ovals represent tasks to be carried out. Stage to stage flow is represented by an arrow connector. We chose to use a containment model and layering, very different from e.g. UML activity diagrams and BPMN process diagrams. We did this as we wanted to allow survey designers to indicate, in a summarised form, key sub-task diagrams, how these are ordered, and key survey artefacts consumed, manipulated and produced. Our survey process diagrams use stages in a similar manner to these other process-oriented visual languages, but incorporate task and artefact relationships.

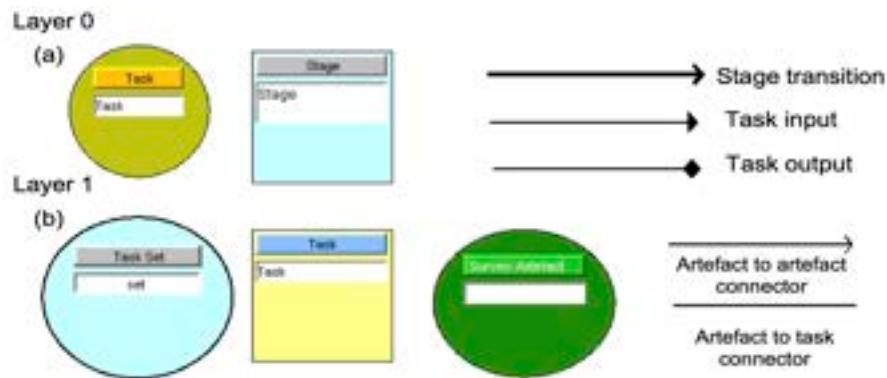


Figure 12. Survey process diagram notation.

A survey process diagram for our crime victimisation example is depicted diagrammatically in Figure 13. This survey process diagram describes two the main stages involved in the survey process: sampling design (stage 1) and data analysis (stage 2). It should be noted that the descriptions for the stages are much simplified and generalised for this paper. Our example involves three tasks: (1) Population study: The population profile is studied to formulate data collection strategies which apply to the sampling design decisions; (2) Sampling design: Sampling method is chosen to meet the required precision of the survey and a trade-off between cost and precision; and (3) Data analysis: To explore the association between the level of education and the crime victimisation. The last two tasks are related to each other in the same task model and the two stages are sequentially linked. This linkage is explored in more detail in the Layer 1 diagram below, which describes the stage tasks in more detail and shows how task(2) provides data for task(3) to consume.

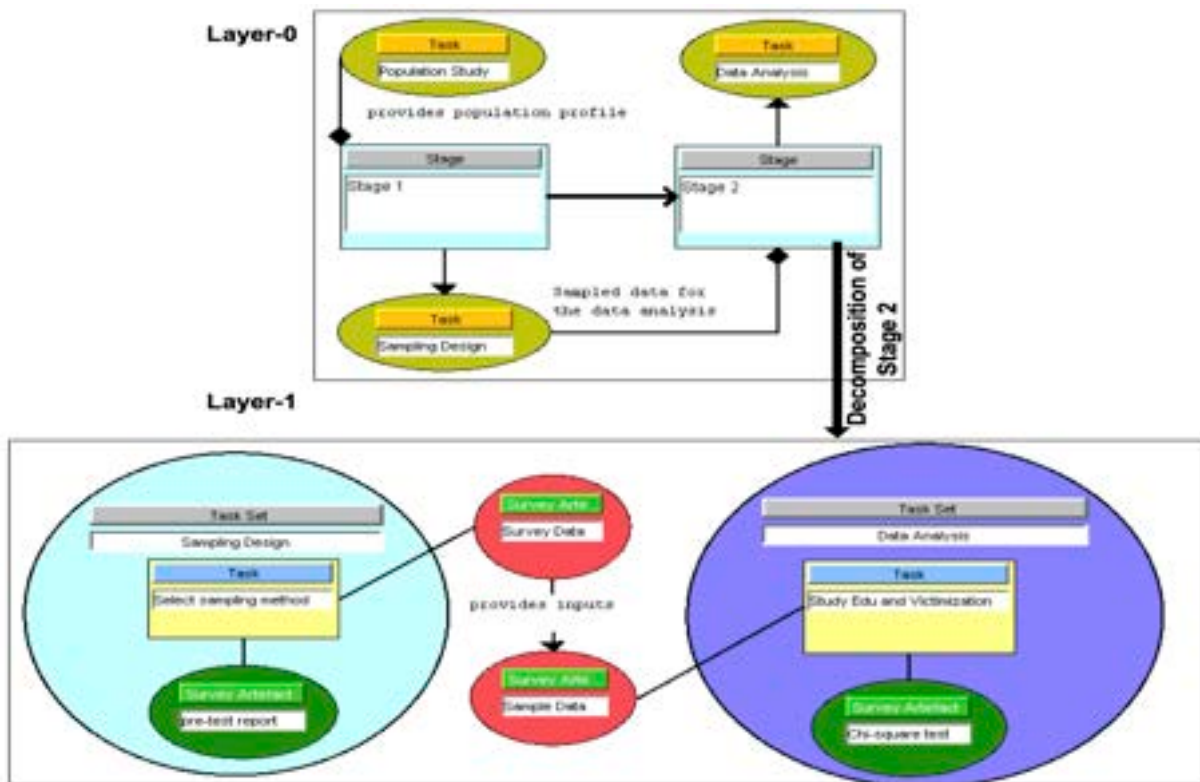


Figure 13. Survey process diagram for the crime victimisation survey.

6. SDLTool Example Usage

We have built a support environment for SDL– called SDLTool. This provides authoring support for SDL, support for importing statistical meta-data, binding data to techniques and data elements, visualising generated statistical data and technique analysis results, generation of reusable web services embodying specified techniques, and a model repository including reuse tools. In this section we use the New Zealand Crime Survey case study to elaborate the steps outlined in Section 4 when using our SDLTool to model the survey, test and visualise survey components, and generate reusable web services for techniques. In subsequent sections we outline the development of SDLTool from conceptual model to implementation.

Initially a statistician designs a survey using the top-level SDL survey diagram, an example of which is shown in Figure 2. The statistician may begin from scratch or reuse a design from an SDLTool repository. For example, in Figure 1, the *Survey* icon plus 4 contexts *Objectives*, *Data Collection*, *Population* and *Data Analysis* form a standard template that has been elaborated with the survey name, an additional context *Caveat*, and survey specific survey attributes for each context.

After creating the overall survey structure the statistician creates further SDL views to specify the survey in more detail. Several examples from the NZ Crime Victimization Survey are shown in Figure 14 (some parts of this survey design have been simplified for the purpose of this paper). Figure 14 (a) is a hierarchical task diagram, specifying two data analysis tasks to be carried out on the survey data. During data collection, our main concern is to specify sampling techniques used in the survey process and types of statistical metadata related to collected data. Figure 14 (b) specifies the two sampling methods to be used in the survey. Here the sampling frame is stratified in two stages by the modified area unit (1) then household visits planned using patterned clustering (2). A mock statistical dataset can be bound to the ‘Data Frame’ icon for pre-testing purposes. Post data collection, collected data can be bound to the ‘Sample’ data icon. Figure 14 (c) shows a survey technique diagram describing the data analysis operations implementing the tasks in Figure 14 (a). Here, we use two visualisation methods (boxplot and multivariate analysis) to assess whether there is evidence of a statistical association between data variables.

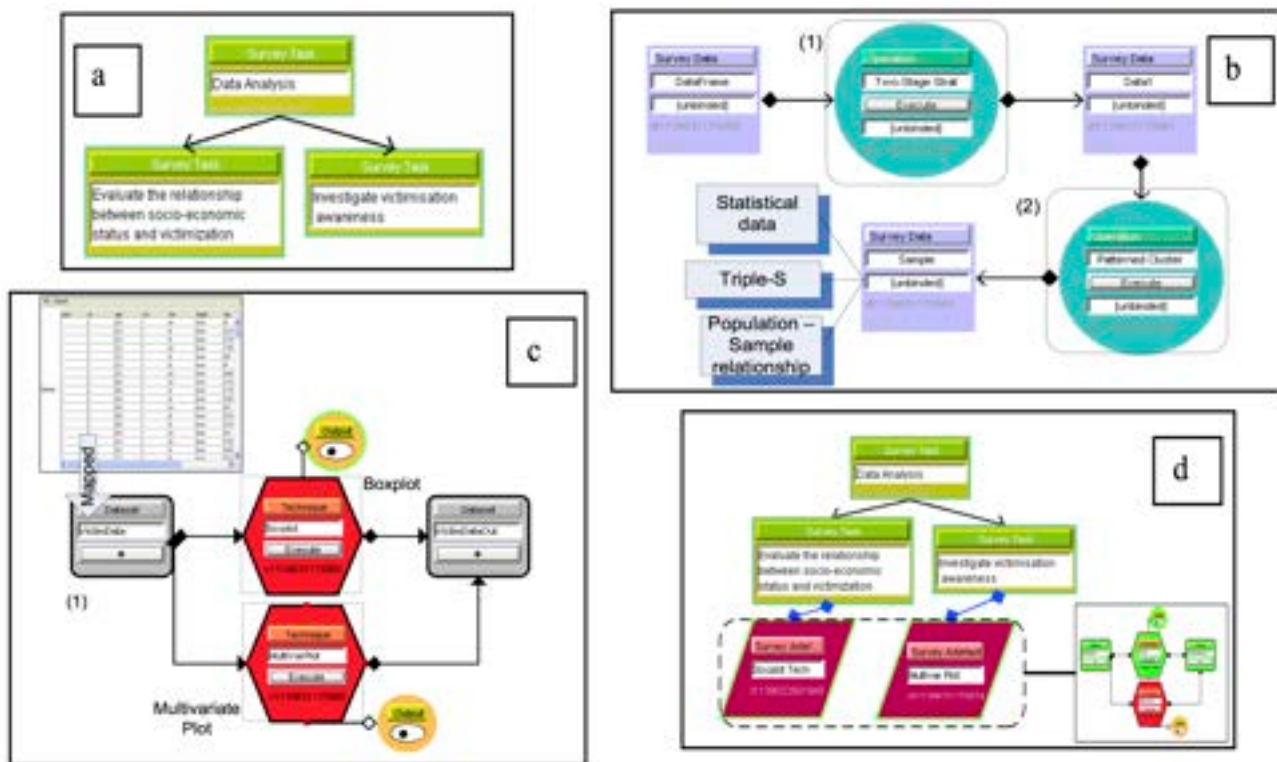


Figure 14. Examples of NZ Crime Survey SDL diagrams.

The two techniques are composed in the same diagram as they use the same statistical data. They are bound to external analysis methods implemented using R. The diagram may then be separately executed to produce a boxplot for the socio economic status and victimisation relationship and the multivariate plot to visualise the public awareness. The modeller also binds the resulting visualisation artefacts to the tasks they correspond to in the task diagram (Figure 14(d)).

6.1 Binding artefacts

Binding operations map SDL icons to various external resources (Figure 15). Menus associated with graphical icons (a,b) initiate resource mappings (dataset and technique respectively). Mapping forms are generated for the user to complete (c,d), with many fields automatically filled, inferred from dataflow and other bindings (maintaining inter diagram consistency). Aspects of the icon related to the mapping appear in separate view tabs where the user can view or edit the resource, eg a data structure (e), data set (f) or metadata description (g). A fully mapped SDL icon has a thicker border indicating readiness to execute.

6.2 Execution

When a survey data or technique diagram is mapped to all required statistical resources the user can execute it in real-time and explore its textual and visual outcomes. Figure 16 shows a typical diagram execution sequence. A completed survey technique diagram has all required graphical icons mapped to appropriate resources, indicated by the thicker icon borders (a). Each technique or data operation icon has a button interface used to execute the specification (with icon colour change indicating successful execution). Following execution, the user may select the output port of the executed technique or operation and probe into available outcomes. Figure 16 (b) shows detailed results of running the multiple linear regression technique on the dataset when double-clicking on the top technique diagram output icon. Modifying the dataset or technique parameters and re-running will update the results shown. Figure 16 (c-f) show various SDLTool built-in visualization support techniques (c-e) and external visualization tools (f) invoked on the output of the MRPairPlot technique. The survey designer can select the desired visualization technique via pop-up menu by right-clicking on the lower output icon in the technique diagram. The visualization technique/tool is invoked and results displayed. Updating the source dataset and/or technique parameters result in internal SDLTool visualisations being immediately updated. External tools are re-invoked and their generated visualisations shown on SDLTool user request. Documentation is also important in managing and reusing the survey process. SDLTool can generate documentation in a form accessible to third-party tools or clients allowing them to understand the semantics of our visual diagrams without the need for the prototype tool. Figure 16 (g, h) show generated HTML documents describing a dataset and statistical technique respectively, as specified in Figure 16 (a).

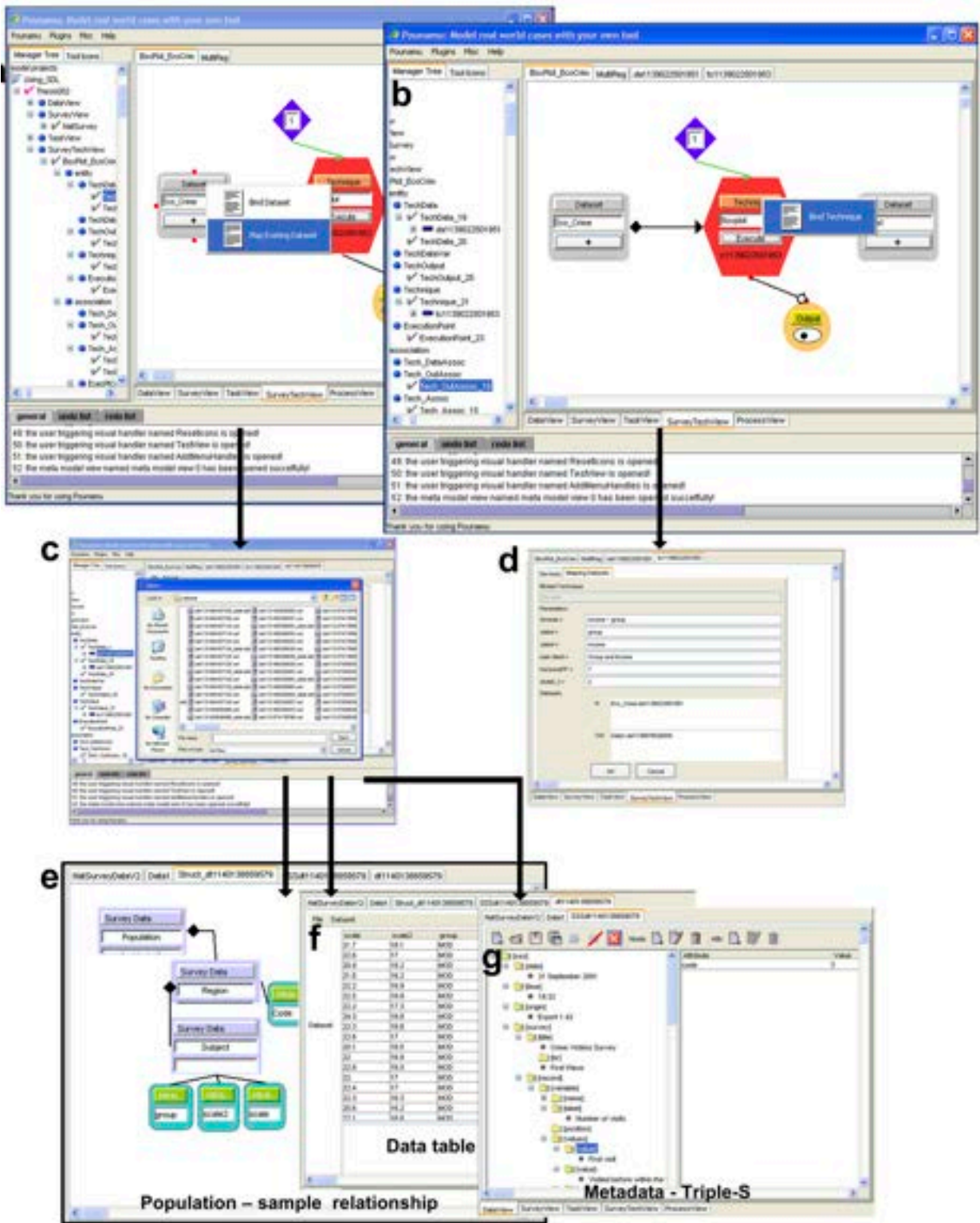


Figure 15: Binding statistical resources to SDL elements

(© 2007 IEEE. Reprinted, with permission, from [22])

All of these activities can be done from within the SDL Tool environment. This thus provides an integrated design, coordination, and implementation environment for complex statistical survey design. The environment is live, providing good progressive evaluation support, with the ability to try out techniques and operations as they are defined and bound to appropriate resources.

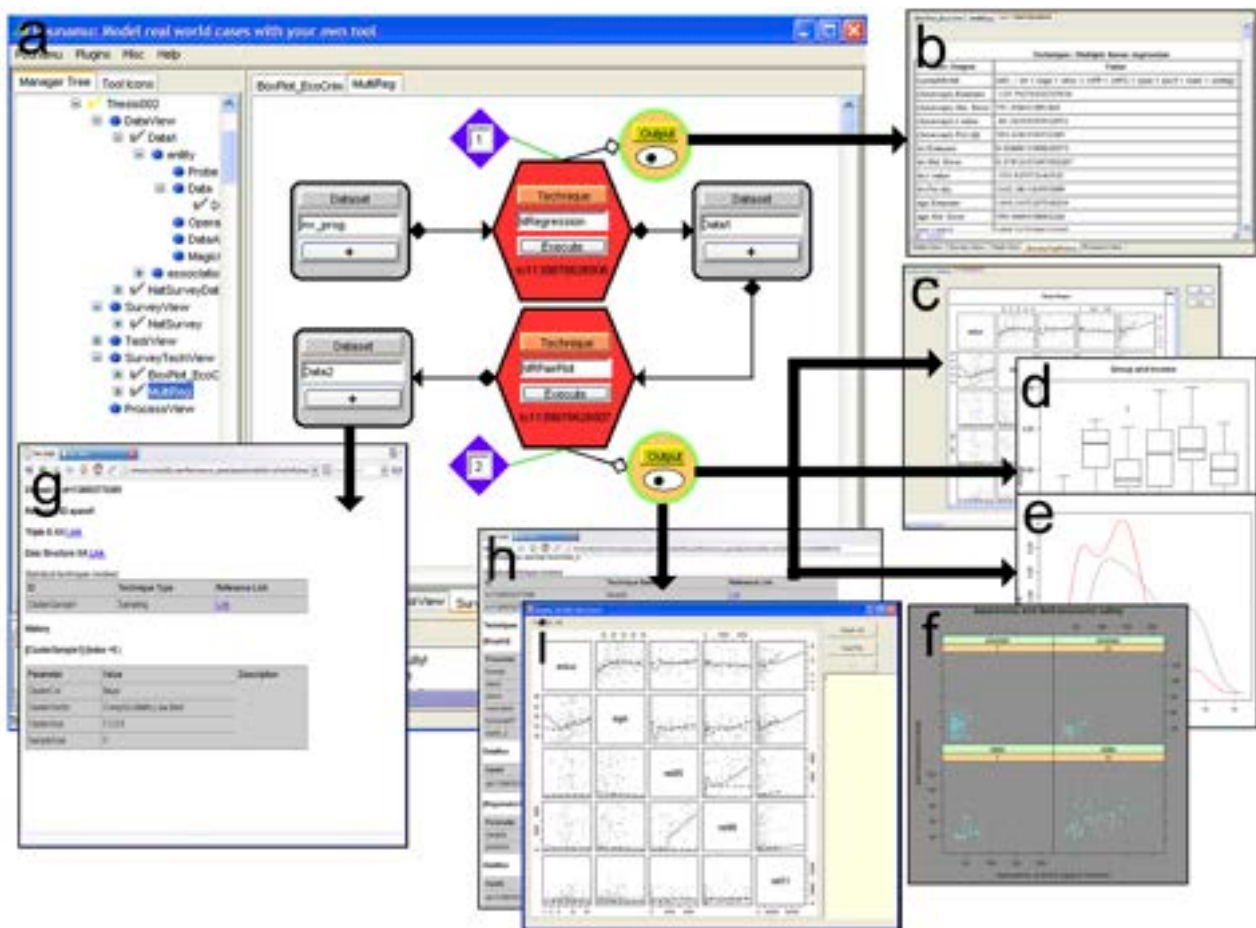


Figure 16: Examples of execution of statistical technique diagrams including results visualization

(© 2007 IEEE. Reprinted, with permission, from [22])

6.3 Web Service Generation

When the user is satisfied with the correctness of a functional technique diagram the diagram can be turned into Java code and exposed in the form of a web service. This eliminates the need to have the SDLTool environment and associated 3rd party statistical analysis packages in order to understand the survey and promulgate its results. To do this, a web service generation template is used to process technique diagrams with generated code stored in a web services repository. Clients can access the service specification via a WSDL interface. Exposure as a web service provides platform independent access to survey results. For example, Figure 16 (i) shows a .NET program using the java-implemented statistical technique of Figure 16 (a) via its web service interface.

The first step in implementing technique web service generation is to create a diagram processing sequence. The implementation of the process examines the underlying diagram structure for execution points according to the diagram's syntactic rules. Once the sequence is complete, metamodel files bound to visual icons are accessed by the service specification generator to create web service specifications. The specifications file encapsulates (i) Dataflows; Techniques (identified by unique identifiers and types); and Parameters assigned to techniques. Once the specification file is ready, the code to execute the techniques according to the defined sequence in the specification is generated. The generation process starts with a technique model definition that describes the data flow from one technique to another, types of techniques used, expected outputs, etc. Technique entities play a central role in code generation. Each technique is evaluated in the order of its assigned execution index starting from 1 and then the code template is used to build up code in Java. The code generation process entails no complex operations due to the autonomous nature of all techniques present in the survey technique diagram. Thus a simple template driven sequence can be applied to the first technique until the last one.

Figure 17 (a) shows the user invoking the contextual menu to generate a service based on the diagrammatic specification of the current diagram. The diagram is submitted to the SDLTool model repository as an XML document as in Figure 17 (b). The submitted model is processed according to a web service generation template and clients may access the generated web service via a generated WSDL interface, shown in Figure 17 (c). Figure 17 (d) shows a demonstration analysis and visualisation application that we built using the Microsoft .NET framework utilising the statistical technique via its web service developed and generated using our SDLTool.

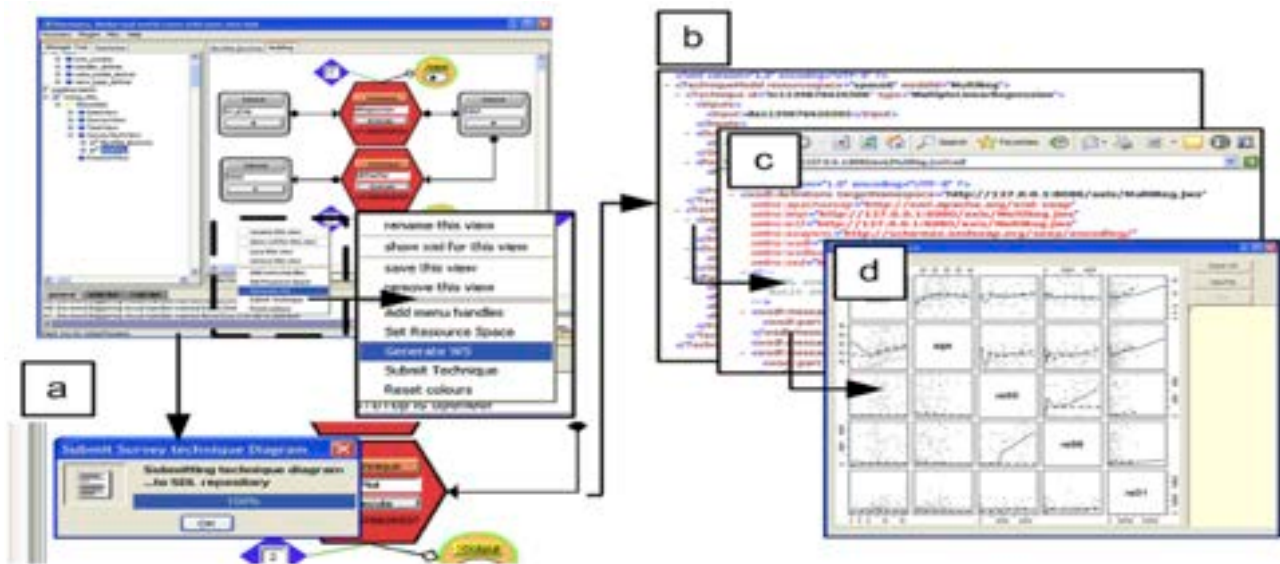


Figure 17. Generating and using a statistical survey technique implementation and its web service interface
 (© 2007 IEEE. Reprinted, with permission, from [22])

7. SDLTool Meta-model and Semantic Layer

As a first step towards realising SDLTool, we developed a detailed SDL meta-model [20]. This was done by a process of abstraction, classification, and generalisation on the collection of SDL visual models in a similar fashion to Model-Driven Architecture [30]. To integrate the SDL model as expressed by multiple SDL diagram types we needed to understand the relationship between elements in each diagram. Figure 4 showed the high level relationship between the diagrams. At a finer grained level, our approach to integrate the five diagram meta-models uses two main constructs: inter-diagram relationships and survey entities. Inter-diagram relationships lay down a broad inter-diagram network and survey entities belong to the network of the diagrams. A survey entity can be defined as follows:

- A survey entity can be a survey task, dataset, technique or non-connector graphical entity in the visual environment.
- A survey entity has its own unique identity though they may appear non-distinguishable visually. E.g. two dataset icons look the same but they are mapped to two unique survey entities.
- A survey entity belongs to at least one visual model.
- Inter-diagram relationships positively imply the existence of at least one survey entity that is shared by more than one diagram. In other words, a shared survey entity completes an inter-diagram relationship.

Survey entities thus provide the basis for both model integration (in the meta modelling process) and inter diagram consistency (at an operational level). The overlapping survey entities act as integration points to merge related views for a target domain, as visualised by Venn diagrams in Figure 18 (left). Users or tools can transverse the network of related diagrams by using overlapping survey entities as entry/exit points.

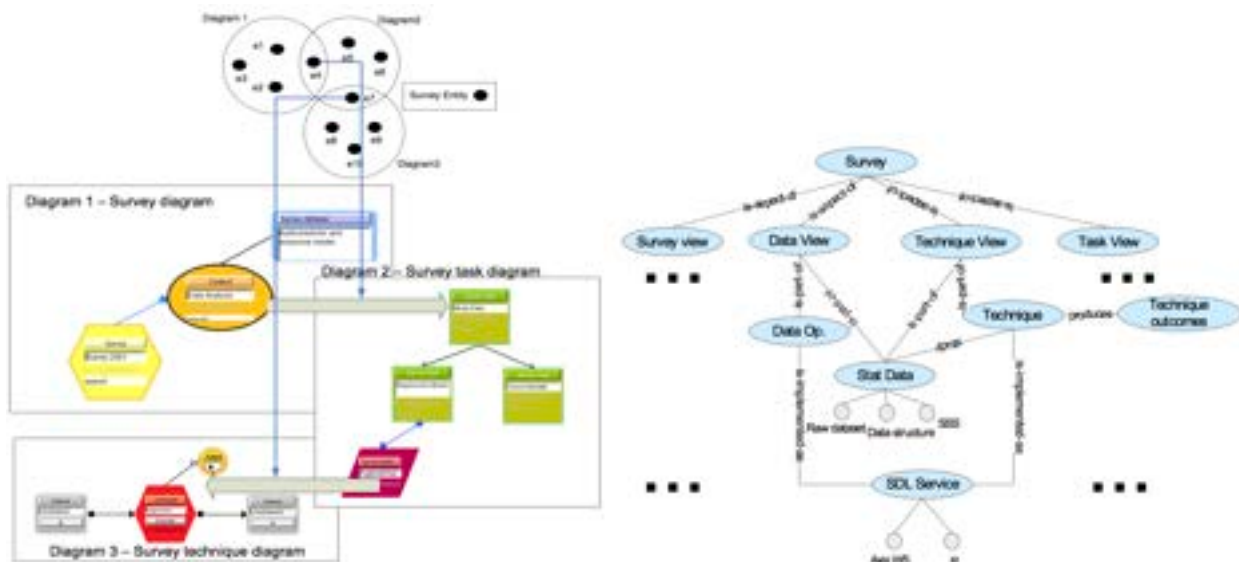


Figure 18: (left) Inter diagram mappings and (right) partial topic-map representation of SDL
 (© 2007 IEEE. Reprinted, with permission, from [22])

The necessity for an additional semantic layer comes from SDL tool support. SDL diagrams as communication media from the perspective of human users require no explicit semantic support. However SDL diagrammatic notations anticipated the development of supporting tools and a model-based approach for generating services. Our current proof-of-concept tools do not fully utilise the semantic layer, relying instead on static rule based reasoning to infer model semantics from the metamodel layer. However the semantic layer will be the primary tool for extending the language base of SDL thus we include a brief description here for completeness.

The SDL semantic layer is primarily organised by *topic maps* [37]. Topic maps offer heterogeneous information repositories [3] to tie the underlying semantic of the metamodel to real world statistical survey topics. The primary constructs in SDL diagrams such as dataset entities are organised into topics and they are explicitly related to the overall conceptual structure of SDL by means of association and instance membership (see Figure 18 (right)). The ontology layer consists of taxonomies of statistical techniques, metamodel structures and relational templates to bind the metamodel to the semantic layer.

If all important aspects and diagrammatic notations of SDL are considered to be “topics”, SDL diagrams are “occurrences” and all inter-diagram relationships are “associations” then we can visualise topic maps as providing a unique platform to express the semantic layer. It is interesting to note that just as we can transverse from the visual layer to the semantic layer, reversing the process by having the visualisation of the topic map as a starting point could potentially be used as the basis for making SDL an extremely extensible visual language.

The semantic layer creates a structural ontology for SDL. Hence the structural ontology also provides a template for which mapping operations to bind external resources (occurrences) with SDL meta-model entities. One such mapping operation is to turn static visual icons into dynamic ones, that is to give the icons the behavioural and functional nature of a widget, and which can then serve as a dynamic interface to control external resources. The icon-to-widget mappings for SDL tool support arise out of the need to support occurrences associated with a topic node at the tool level. Thus the semantic layer also provides a new perspective in looking at visual tool support from the modelled ontology.

8. Architecture and Implementation

Our SDL tools use an event-driven, loosely-coupled architecture as outlined in Figure 19. At left, SDL diagrams are represented as diagram data (or “views”, diagram left) and a shared repository (or “model”, diagram middle) following the Model-View-Controller paradigm. A set of extensible components (diagram right) are used to provide repository support, external tool integration, model compilation, execution engine, and a web service generator to make SDL designs accessible to other users. Brief descriptions of each of these components can also be seen (table right).

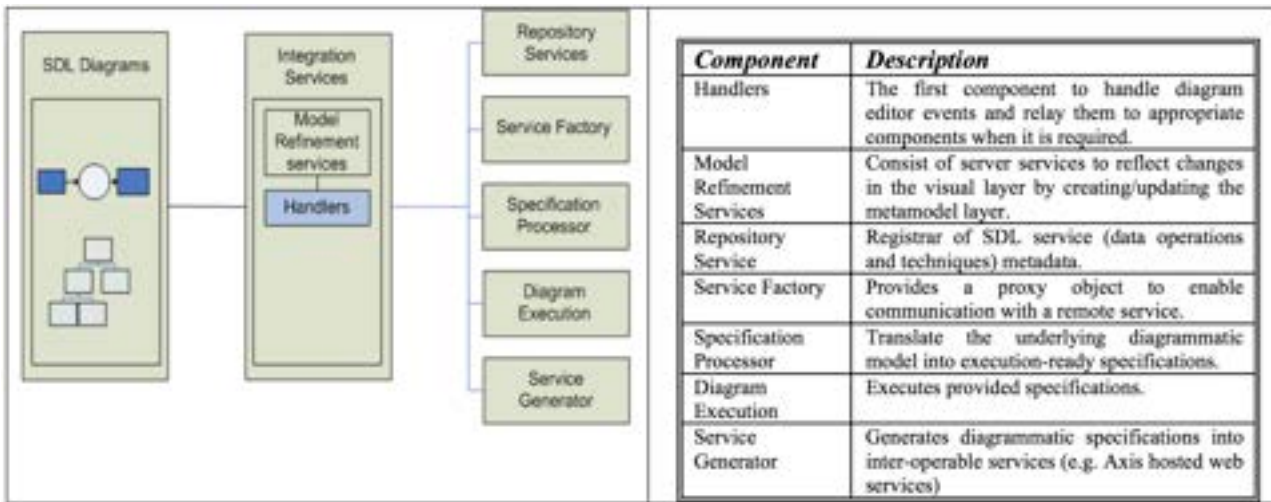


Figure 19: SDL tool architecture

(© 2007 IEEE. Reprinted, with permission, from [22])

Figure 20 shows how events are used to couple the components in our architecture. Changes to diagrams are sent as event notifications to an “event handler” for the diagram (1). This passes the event onto a model refinement service (2), which propagates the event to components subscribing to the diagram change type e.g. the Repository Service. The Repository Service translates SDL data into the dataset file format and updates this (3). A response event is generated by the service to indicate success or failure. This event is processed by the model refinement service (4) to determine if any updates to the shared SDL model are necessary. If so, these are applied to the model (5). Such changes may mean other SDL diagrams sharing the changed model data need updating (6). This event-based notification mechanism provides incremental multi-view consistency, persistent repository support, compilation of SDL models, an execution engine for diagrams, and an external tool integration platform.

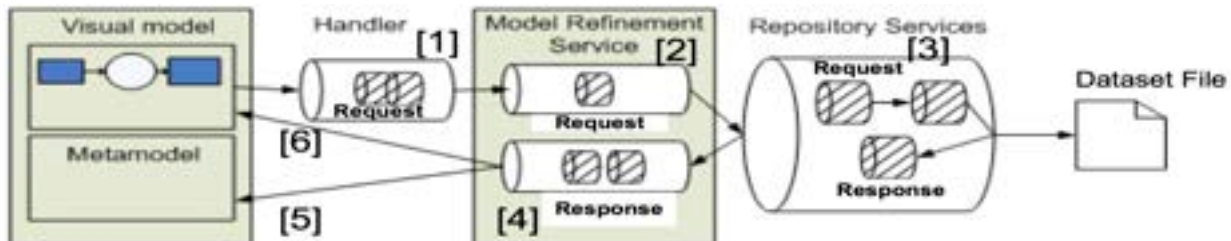


Figure 20. SDL toolset analysis pipe-line

(© 2007 IEEE. Reprinted, with permission, from [22])

We used our Pounamu meta-tool [46] to implement our SDL multiple-view design tools. All of the SDL diagrams in this paper were generated from screen dumps from these editors. Pounamu allows multi-view, multi-notation diagramming tools to be quickly specified using meta-model, view type, shape definer and event definer meta-tools. Its support for “on the fly” changes to tools allowed us to readily experiment with notational elements and diagram types and modify them at low cost. SDL was specified as a canonical Pounamu meta-model and a set of view types, one for each different SDL visual notation. Each view type (diagramming specification) has its own set of shapes, connectors and editing constraints. The SDL diagram editors are realised by Pounamu interpreting the SDL tool specifications to provide multi-view and multi-user diagram editors with a shared model. The Pounamu model produced by the SDL diagramming tools is used to provide a data-oriented integration platform to external statistical analysis tools. The Pounamu SDL shared model is “walked” by Pounamu “event handlers” developed specifically for each external tool, transforming it into a format understood by that tool and the tool is invoked via service factory components. Handlers also provide presentation and control integration for external tools allowing results to be displayed in the SDL tool.

Figure 21 shows a more detailed set of components making up the SDLTool architecture. When using SDLTool, a meta-tool specification is opened by Pounamu (a) and SDLTool is instantiated. Statisticians build SDL models in SDLTool, having diagram and model information stored locally (b). SDLTool uses an event-driven (c) generator to turn technique diagrams into scripts to drive external 3rd party statistical analysis tools (d). SDLTool may invoke these

scripts directly with example data sets, allowing testing of a technique implementation (e). The technique web service generator produces a web service providing an interface to the generated scripts for the 3rd party tool analysis and a WSDL specification for this interface (f). The generated technique web services are deployed (g) for remote discovery and invocation. 3rd party client applications may (h) discover and invoke (i) the generated technique web service to make use of its statistical survey technique implementation (j). Datasets are passed via the web service to the 3rd party statistical analysis packages (k), processed, and results returned (l) to the invoking 3rd party applications.

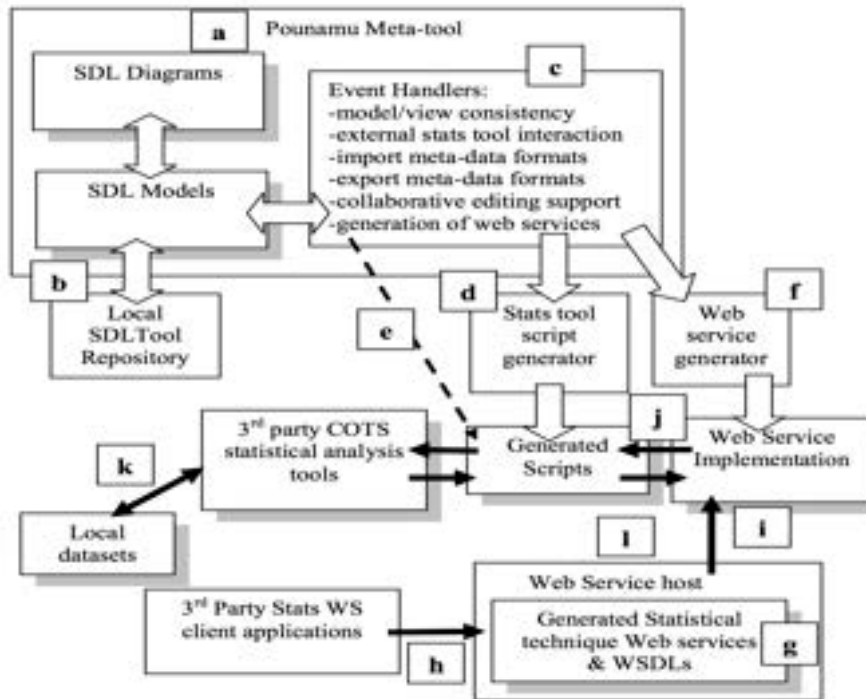


Figure 21. SDLTool architecture (© 2007 IEEE. Reprinted, with permission, from [22]).

The script generator takes a survey technique diagram and generates one or more scripts for 3rd party statistical analysis packages like R. These are “technique implementations” that given suitable input datasets will clean, select, analyse and produce an output datum or dataset for the specified technique algorithms. The web service implementation generator produces a Java implementation of the web service that is used to receive datasets as XML messages, transform them into local data files in appropriate 3rd party tool proprietary data formats, invoke the 3rd party statistical analysis package with appropriate generated technique script implementation, and transform any result data from the 3rd party tool’s proprietary format and back into an XML message.

9. Evaluation

Recall our key research question from Section 2:

RQ1: Can visual notations with appropriate software tool support be used by end users, i.e. statisticians and other survey developers, to facilitate statistical survey design and implementation?

To help answer this question we undertook two cognitive walkthrough-based [17] evaluations, primarily targeted at the “design” aspect of RQ1, and a third evaluation where we designed and built several statistical surveying techniques with SDLTool and generated web services to provide access to these, targeted at the “implementation” aspect of RQ1. We evaluated the services by using them within other statistical surveys. For the two cognitive walkthrough studies the theoretical testing basis for the evaluation was the cognitive dimensions framework [16]. In the following, italics are used in the text body to highlight specific cognitive dimensions.

For the initial cognitive walkthrough we used a doctoral student in statistics as our test subject. The aim here was to provide a preliminary answer to RQ1 with a near final version of toolset to help shape a final iteration of the language and tool design and to act as a pilot to help design the subsequent evaluations. This user was an expert end user of statistical survey designs and techniques.

The second cognitive walkthrough evaluation provided a more formal usability assessment of SDLTool using eight test subjects, some very knowledgeable in statistics and some with basic statistical surveying and analysis knowledge.

9.1 Pilot cognitive walkthrough

Methodology

In the initial cognitive walkthrough study we observed a doctoral student in statistics making use of SDL and SDLTool to design a moderately complex statistical survey. We chose to use such a test subject as they are very knowledgeable about existing statistical processes and tools, are an expert user of the range of features of SDL and SDLTool, and could give us detailed and expert feedback during each step of the walkthrough on their thinking and tasks. We used an earlier version of SDL and SDLTool than described in this paper for this cognitive walkthrough, with findings from this leading to several visual language and tool enhancements.

Our test plan for examining SDL using such a cognitive walkthrough approach comprised the following elements:

- Overview of SDL - the test subject was briefly introduced to SDL and working examples explained to observe SDL in action.
- Users Tasks - a list of tasks to be performed by the test subject was given. As the cognitive walkthrough approach focuses on user-oriented solution finding, the tasks were intended to give the test subject opportunities for a self-initiated exploratory path to complete the tasks. Thus the tasks attempted to simulate the cognitive context of a survey researcher in practice rather than imposing fine-grained questions. Three tasks were given to the test subject, all designed to model real-world problems. The first was to request the subject to design a survey diagram for a large-scale UK government sponsored labour force survey [34]. In the second task the subject was given a survey data diagram and requested to explain it and comment on the information represented. The third task involved the subject designing a survey of his choice from scratch.
- User Awareness- a well-designed visual language should give users the sense of self-awareness. In other words, users should be able to tell whether they are on the right track in terms of meeting final goals during the course of the using SDL. Insufficient user awareness can especially impact the usability of the diagrams, which are affected greatly by local changes, as late changes could imply a significant overhaul. Therefore the evaluation of SDL looked into not only the final results but also user awareness throughout the testing session.

Results

The test subject successfully composed a survey diagram, shown in Figure 22 (left) after a brief introduction and with no interventions. The subject quickly identified survey contexts and added the key survey attributes to each context. The subject found the notation intuitive and easy to use. Only one small fault was identified in the diagram produced. The attribute in the bottom right corner should have been directly connected to the *Subjects* context rather than to the other attribute as it is not an extension of that attribute.

For Task 2, our subject was initially given the survey data diagram of Figure 22 (right) This used an early version of the notation where sampling operations were linked back to the original data structure. From a data type perspective this makes sense, but was found to be confusing by the test subject. This led to the revision shown in Fig. 5, where operations on a data structure results in a new data structure, which should be less confusing for non programmers. However, we have retained the self-reference type representation as a convenient shorthand for complex diagrams with the knowledge that this abstraction requires significant learning.

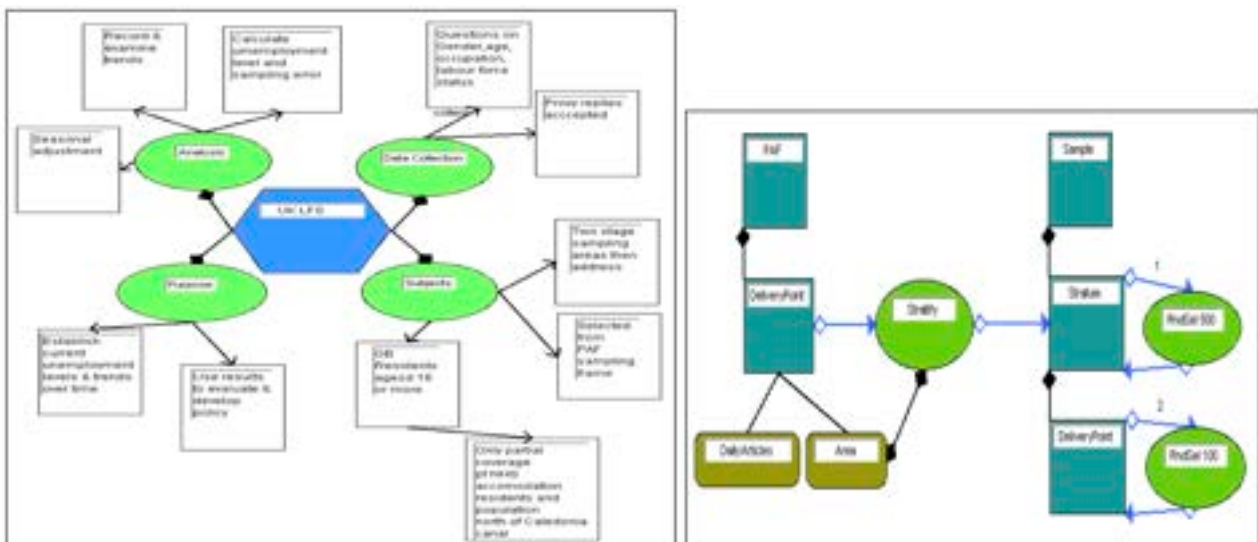


Figure 22. (Left) Survey diagram generated by test subject and (right) Initial survey data diagram given to subject

(© 2005 IEEE. Reprinted, with permission, from [23])

After the first two tasks our subject was asked to derive a survey design from scratch. No specific guidelines were set. The subject chose an analytical survey design based on a harvest estimation survey that he regarded as typical of analytical agricultural surveys. The core operational details of the survey were discussed and immediately those details described in SDL diagrams. The survey process included many iterative decision-making components, but our subject found these to be readily represented in SDL. Figure 23 shows the survey process diagram created during the test session. The diagram demonstrates the advantages of SDL in easily turning large amounts of survey design information into a comprehensible visual model that is both clear and has explicit semantics.

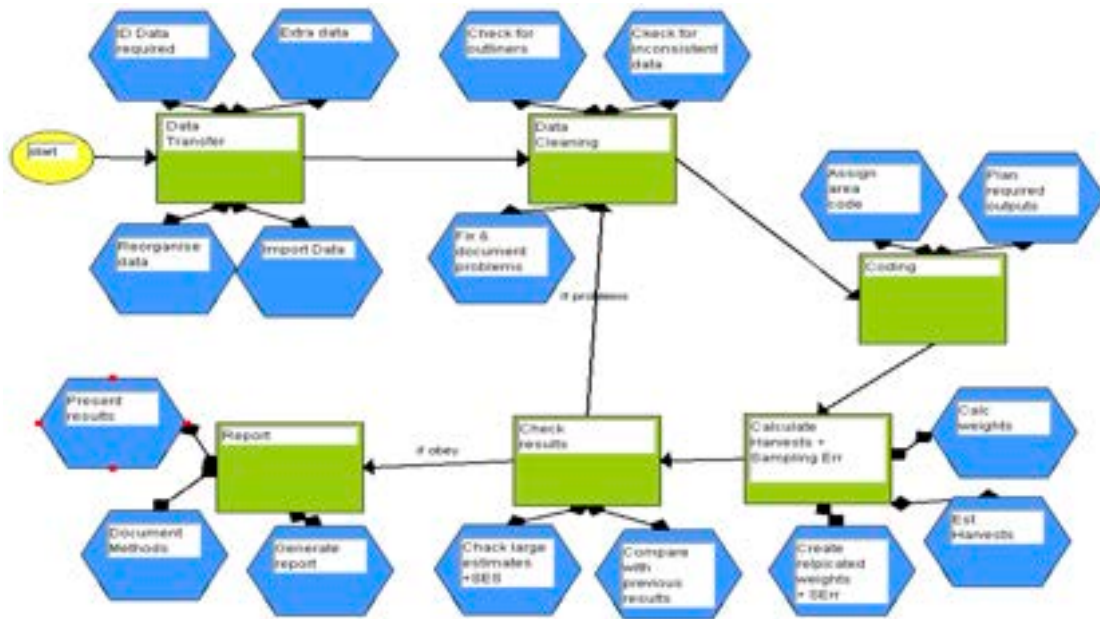


Figure 23. Survey process diagram generated by test subject for harvest estimation survey

(© 2005 IEEE. Reprinted, with permission, from [23])

Interpretation

The test subject readily and capably applied SDL for modelling the survey in a short space of time and was able to follow rules for association and visual symbols, not repeating the earlier error of misplacing survey attributes supporting an affirmative answer to RQ1. SDL provided a compact description of the survey design describing the activities, including events or items that are required in the execution of the survey process, and associations and revealed the overall purpose and structure of the survey. SDL has a number of limitations identified by the test subject. All of these originated from diagram layout concerns. One example is the survey process diagram’s tendency to spread out over a large space. It can be more than an aesthetic problem as the spread-out look may slow down a user’s comprehension by presenting more visual components than a single visual scan can perceive. One quick solution to resolve the visibility problems is to use multiple views, with each view elaborating on only a limited number of contexts. However, this introduces *hidden dependencies* and *juxtaposition* issues. Another would be making the processes elidable and to build an intelligent layout algorithm to shape the entire diagram to optimise visibility. Future work with Pounamu, used to build SDLTool, has included some improved reusable layout support, motivated by SDL and other DSVL layout problems.

9.2 Formal User Evaluation

To investigate RQ1 from a target end user’s perspective, we carried out a further study with 3rd Year Statistics student participants. The testing session incorporated a cognitive walkthrough approach to gather qualitative data [17], together with a survey instrument to obtain user perceptions. Task completion data were also gathered.

Methodology

Test subjects were recruited from final year undergraduate statistics students at the University of Auckland. They were chosen out of a potential candidate pool resulting from recommendations from a tutor in our Department of Statistics.

All test subjects were invited to attend an introductory meeting that was designed to convey some of SDL's core concepts and to provide a basic tutorial on SDL and SDLTool.

A wide range of user activities were conducted in each testing session: pre- and post-demonstration interviews; diagram comprehension exercises; survey technique implementations using provided software tools; and comparative evaluations of SDL and existing statistical survey support tools. Two types of testing outcome were compiled: user-completed questionnaires including user perceptions, opinions and satisfaction regarding the SDL tools and their own performance; and investigator recorded performance evaluations, including task correctness, mistakes, and time to complete given tasks. We ranked performance of each task and subtask using a combination of these measures.

The key tasks asked of subjects were:

- Task 1 – SDL and SDLTool preview and tutorial (instructor led);
- Task 2 – SDL diagram comprehension: survey diagram, survey task diagram, survey data diagram, survey technique diagram and mapping of SDL diagram components to external data and services;
- Task 3 – SDL diagram and service construction: survey diagram, task diagram, survey technique diagrams; web service generation and testing

The questions asked of subjects were:

- Is it easy to model the survey with the tool?
- What does the tool do well in your opinion?
- Are there any notations that should be made clearer for the user? How? (e.g. more user interaction behaviours, colour codes)
- Is there anything the tool does not let you do that you would like to?
- What other information or views of information should the tool display?
- Are there any improvements you would like to see in the tool?
- Would you like to use the notations if you had a large survey project to do? Why/Why not?
- Are there any concepts/ideas in the SDL diagrams that are difficult to comprehend?
- Is there anything the diagram does not let you understand that you would like to?
- What other information or views of information should the diagram convey?
- Are there any improvements you would like to see in the diagram?
- Any general comments you have on the concepts of the tool and notations.

The introductory SDL diagrams used in this experiment were based on the 2001 New Zealand Crime Victims Survey to simulate real-life survey communication problems. Test subjects were asked to explain the semantics of presented diagrams and to give feedback on their effectiveness, expressiveness, usefulness and usability. Activities utilizing the software tools were to implement survey techniques to produce solutions for given Crime Victims Survey scenarios. The scenarios were designed to permit user-initiated actions and task execution.

Results

Eight test subjects, all final year undergraduate statistics students at the University of Auckland, participated in the user testing. All were new to the concept of a visual environment for statistical surveys but all had good working knowledge of statistical packages such as R and SAS, survey theory and design in academic or commercial settings. Three participants were competent programmers of general purpose high-level languages such as Java and C

Figure 24 shows the results of Task 2 – diagram comprehension. Performance of each task was ranked by a combination of user-reported feedback and experimenter (the first author) observation. Figure 25 shows the results of Task 3 – diagram and service construction. Performance of each task was ranked by a combination of user-reported feedback and experimenter (the first author) observation.

Task 2 - Diagram comprehension

#	Survey Diagram	Task	Data	Technique	mappings	Average	Rounded
1	2	3	2	2	2	2.2	2
2	3	3	3	3	3	3	3
3	3	2	3	3	2	2.6	3
4	1	1	2	3	1	1.6	2
5	3	3	3	3	2	2.8	3
6	1	1	2	3	1	1.6	1
7	3	3	3	3	3	3	3
8	3	2	3	3	3	2.8	3
Average by diagram		2.375	2.25	2.625	2.875	2.125	

0=Failed, 1=Average, 2=Good, 3=Excellent

Survey Diagram

Survey Diagram

Task

Survey Task Diagram

Data

Survey Data Diagram

Technique

Survey Technique Diagram

mappings

Diagram mappings and SDL tools

Figure 24: Task 2 results summarised.

Task 3 – Diagram Construction

Participant #	Task1	Task 2	Task 3	Task 4	Average	Rounded
1	1	1	1	0	0.75	1
2	3	3	3	3	3	3
3	3	3	3	3	3	3
4	2	2	2	1	1.75	2
5	2	2	2	1	1.75	2
6	0	1	0	0	0.25	0
7	3	3	3	2	2.75	3
8	2	2	2	1	1.75	2
Task Average		2	2.125	2	1.375	1.875

0=Failed, 1=Average, 2=Good, 3=Excellent

Figure 25: Task 3 results summarised.

Figure 26 summarises the results of analysing user responses to the survey questionnaire and the task completion data. The former are summarised in the first two categories where qualitative responses were analysed and categorised into positive or negative opinions. The latter are summarised in the last two categories, which were categorised according to how complete the solutions provided were against pre-set criteria. Combined these provide general quantitative indications of the usability and efficacy of SDL and the SDLTool.

Category	Results
General tool usability	Positive 87.5% Negative 12.5%
Overall notation usability	Positive 75% Negative 25% (half these were only partial negative)
Diagram comprehension (user performance)	Excellent 62.5% Good 25% Average 12.5% Incomplete 0%
Task completion (user performance)	Excellent 37.5% Good 37.5% Average 12.5% Incomplete 12.5%

Figure 26: Results of user survey

(© 2007 IEEE. Reprinted, with permission, from [22])

Even though the test subjects were new to the concept of using visual language for statistical surveys, the subjects were able to understand and use diagrammatic notations to express various aspects of the survey process and compose statistical techniques to solve test scenarios. The learning curves of the subjects varied but all of them were able to comprehend testing diagrams to the level required for their tasks.

Interpretation

Key findings from the detailed survey comments and observations include the following:

1. SDL accentuates and integrates the multiple aspects of a survey that are often not addressed by existing practice. This makes SDL based solutions more user-oriented. Users interacted with visual notations not just to formulate numerical computations but also to convey actual operational semantics behind those activities. The test subjects responded that this approach would scale well to communicate large-scale survey projects.
2. Participants reported that the visual modeling approach and inter-diagram mappings helped users to think of a survey project as set of tasks within the context of the whole survey process and individual survey constructs to be conceptualized as reusable components.
3. The effect of the visual approach on comprehension performance of both high and low level details of the survey process varied. Each diagram drew different responses but with overwhelmingly positive overall feedback. Survey diagrams were received well by the test subjects as being easy-to-use, expressive and a time-effective alternative to conventional documentation. Two participants viewed survey task diagrams as too radical a departure from existing practices, but the majority stated that the diagrams visualized a valuable aspect of the process. It was interesting to note that the three participants who had previous experience in highly specialized statistical research roles were the most enthusiastic about the concept of task models. Their professional or academic backgrounds ranged from financial modelling to accounting. The tacit existence of prevalent patterns in those specialized fields seemed to be largely responsible for their reactions. Both survey technique and data diagrams were received favourably. They shared strengths and weaknesses as they look alike and convey information at similar levels. Their most notable strength was in capturing an integrated view of sampling, statistical metadata and techniques using an intuitive dataflow metaphor. Negative feedback mostly originated from a lack of *secondary notation* support.
4. As visual language novices, it was not a trivial task for participants to visualize inter-diagram relationships and harmonize disparate diagrams into a single unified model. Even though test subjects did very well in comprehending and utilizing individual diagrams they expressed some difficulty in mapping all the diagrams together when designing the survey process due to *hidden dependencies*. This problem is analogous to a novice UML user's difficulty in merging all UML diagrams mentally together to form a unified view and resulted in some cases in the introduction of inconsistencies. One promising solution suggested by the test subjects to remedy this design issue was the creation of a visual layer to dynamically illustrate how the underlying model is formed by contributing diagrams and the relationships amongst them. Future development of SDL will explore the feasibility of this approach.
5. All five diagrams allowed all graphical entities to be viewed and manipulated in a single view pane at the individual level. This may have addressed a significant portion of *visibility* criteria but without automatic layout management to beautify graphical constructs, visual distractions accumulated as the size of a diagram grew. A diagram layout management component (e.g. automated tree layout management) would alleviate the visual distraction level of large-scale diagrams.
6. One important aspect of SDL is its ability to have a survey represented in multiple diagrams and an SDL diagram's graphical entity may have several sub-level layers that edify lower level information associated with the graphical entity. This design feature of SDL could contribute to potential *juxtaposability* problems, which were commented on by participants. For example a survey data diagram's data entity may have up to 3 sub layers associated with the entity. The presence of the associations is visually noted by changes in the entity's boundary line shape and colour. An ad-hoc remedy such as putting lower-level information on one top-level layer might address this issue but would require a heavy trade off in a diagram's level of visual clutter. *Juxtaposability* could be enhanced by offering tool level enhancements. For instance more fluid navigation between diagram layers allows user's working memory to be minimally disrupted by changing scenes. Another tool-level approach to remedy the trade-off could be interactive 3D visualisation of SDL diagram layers. A 3D environment may provide better a visual perspective for users that side-by-side presentation in 2D with the same screen dimensions cannot offer.
7. Participants responded favourably to the degree of freedom afforded by the lack of *premature commitment* and low *viscosity* the SDLTool offers in designing survey processes and the broad range of diagram types supported. No existing specialized survey tools have functionalities corresponding to survey and task diagrams drawing favourable comment from participants. Participants also noted that, unlike existing tools, they were not restricted to a sequential batch mode or an interactive mode, which tends to require *premature commitment* and also high attention investment [4] to articulate a technique that needs to be frequently modified. Existing tools emphasise result oriented step-by-step batch

operations or UI based pre-packaged procedures. SDLTool's target emphasis is not exclusively result oriented. Its diagrammatic notations aim to capture the whole semantics embodied in a statistical technique thus allowing a type of expressiveness that is intuitive to the user. The *low viscosity* of the data flow metaphor utilized in survey data and technique diagrams provided users with design-time freedom to change input and output data flows and the dynamic mapping of a graphical entity to a physical dataset or statistical technique meant data flows could be routed to multiple techniques in a variety of ways to easily form many variations of the initial design model. All these positive aspects made SDL tools successful in providing visual cues for the whole process while also providing direct manipulation interfaces for a wide range of more detailed operations.

8. Developing and implementing statistical techniques in many popular statistical computing packages entails the translation of symbolic mathematical statements into tool specific languages. When operational requirements are embedded into the survey process, users need to switch back and forth between two vastly different modes of mental operation. If implementations in both mental modes are *inconsistent* or demand *hard mental operations*, the whole survey process can induce harmful side effects such as *a high abstraction barrier*, and *viscosity*. In the visual environment supported by the prototype tool, the test subjects were expected to know correct usage of statistical techniques but they were separated from low-level representations of the techniques that only exist in the forms of service components. Technique constructions are metaphoric abstractions that reside in a single domain. Thus a technique composition in actuality is a model building exercise that is entirely independent from underlying platforms. Although these enhancements of the prototype tool brought a new level of usability to the test subjects, which they responded favourably to, some instances where the model-level coupling would work against performance efficiency were also found. For example the current SDLTool does not allow users to do code level modifications; expert participants who were comfortable with raw code modifications in R's interactive mode felt they would outperform the SDL-based approach when the statistical techniques, which are provided as a service to SDL tools, are themselves subject to frequent changes. Therefore we will consider possible tradeoffs in implementing a dual-mode access to mapped services in the future development of SDLTool. The example illustrates how established practices may unexpectedly deteriorate the quality of a solution founded on theoretically presumed user patterns.

10. One of the design principles behind SDL was to minimise the set of graphical entities in a diagram (*low diffuseness*). This design decision was principally made to lower the accessibility of the barrier to accommodate a diverse pool of users. The approach was justified by positive user responses regarding the usability of diagrammatic notations and SDL tools. However the presumption that less equals better usability was not the case across all user experiences. As the participants became more immersed in the visual environment, their demand for more direct visual control of data flow grew in scope and functionality.

Our prototype tool allows users to navigate to a sub layer to modify parameters of a mapped service via a dynamically generated UI but once they leave the layer they are no longer able to edit the mapped service either by indirect or direct manipulation and the values of the parameters are hidden behind a graphical icon. Direct control of the mapped service is delegated to the sub layer primarily to reduce users' memory load and to provide an exact contextual perspective for each visual layer as discussed in one of the design principles of SDL. One of the interesting observations during the testing sessions was that more competent users requested the ability to manipulate mapped services within the top-level diagram without navigating to appropriate sub level diagrams. Further post session discussions revealed that the user request is reminiscent of shortcuts. Shortcuts provide a secondary access to application functionalities in a typical GUI design for users who wish to bypass GUI based interactions to save time and effort. Likewise, visual shortcuts should improve user performance by offering time-saving alternatives. A visual shortcut approach will be investigated in conjunction with better elision support in our future work.

Overall, both the quantitative and qualitative analyses support an affirmative answer to RQ1.

9.3 Statistical Service Generation

To further evaluate the "implementation" dimension of RQ1, we evaluated the web service generation support of SDLTool by using it to develop a range of statistical survey designs and technique implementations for various moderately-sized statistical problems.

Methodology

Our exemplar survey technique web service generation problems came from the New Zealand Crime Victimization Survey and a large existing survey, Predictor of One-Year Development Status in Low Birth Weight Infants [38]. Each of these surveys makes use of a set of tasks, processes, meta-data, data and composed statistical techniques. Generated implementations of survey techniques are a set of scripts, sometimes very complicated, used to drive one or more 3rd party statistical analysis tools. These generated statistical technique implementations are made available to remote, 3rd party applications via generated web service interfaces. Generated WSDL specifications for the web service permit advertising the service via a registry and discovery and invocation of the technique implementation. Generated web

service code supports data translation from remote clients into the target 3rd party tool input format, and from the output format of the 3rd party tools into XML for return.

Surveys rarely become obsolete in the way software does. Thus revisiting existing surveys is common practice for survey developers and users. SDL diagrams can be used to aid review and restructuring of existing surveys by highlighting core semantics of the surveys. We also used SDLTool to model and modify the Predictor of One-Year Development Status in Low Birth Weight Infants [38] case study. This allowed us to investigate a conversion procedure to turn existing survey descriptions and techniques into SDL diagrams and some reusable survey technique web services.

Results

We successfully reused the generated technique web services from within SDLTool, from .NET-implemented remote clients for the statistical survey service, and from scripts used by 3rd party applications. The current tool only turns survey technique diagrams into executable web services, as they are the aspect of survey process design most amenable to code generation. Each technique diagram represents an independent, stateless and reusable set of atomic statistical techniques within the context of a single activity without the orchestration overheads that are inherent in behavioural diagram types such as UML activity diagrams and SDL process diagrams. Generated statistical technique implementations manifested as web services are currently deployed to an Axis server with a single implementation invoking a single 3rd party statistical analysis tool script. Some of these technique implementations are very heavy-weight and simultaneous use of such services by multiple clients is not feasible. A more scalable deployment architecture is required for these technique implementation web services. This includes the ability to deploy multiple services to different web service hosts; the ability to deploy a technique web service and associated generated scripts and 3rd party tools as a unit; and the ability to performance test and engineer the deployed services.

The synthesized statistical technique implementation web services produce proprietary WSDL message formats rather than utilising any standard protocol for encoding statistical data and decoded results data. This means 3rd party client applications using the generated technique web services must be engineered with knowledge of these protocols. With emerging meta-data standards for statistical analysis tools these generated web services should at least conform to an XML-based data representation scheme for such techniques. Additionally, as web service-based technique implementations become more widely accepted in this domain, new protocols to invoke such technique implementation services are likely to emerge. Again our generated web services should conform to such standards to make them compatible with potential 3rd party client applications.

In our experiment with remodelling and reusing the Predictor of One-Year Development Status in Low Birth Weight Infants survey, results were positive. The survey design was readily able to be modelled and some initial design procedures developed to assist survey researchers to convert existing textual survey designs into SDL. We managed to package for reuse several statistical techniques in the survey as web services for reuse by not only a reimplementations of this survey but by other survey tools as well.

9.4 Strengths, Limitations and Lessons Learned

Our three evaluations of SDL and SDLTool have identified their key strengths as being:

- SDL and SDLTool supports all aspects of survey design and development, some better and more completely than others.
- Many aspects of statistic survey design and development that are currently ad-hocly described and carried out are both formalised and structured, especially the inter-relationship of different aspects of survey design.
- The distinct visual representations used for each SDL diagram type are generally accessible and found to be useful by participants in our studies
- The generated web services embodying survey techniques are reusable from SDLTool-based survey designs and by a range of third party survey data analysis tools

Weaknesses that we have identified include:

- While each SDL diagram type is limited in notational elements and semantics, the combination forms a large and relatively complex visual language suite, resulting in a moderate learning curve, similar to that of UML does for software modelling.
- Inter-process dependencies are not particularly well expressed and SDL still has significant hidden dependency issues, a problem common with many DSLs.
- The underlying SDL meta-model and composite model formed by the multiple SDL diagrams constructed in SDLTool is hidden from the user; making these composite elements and properties accessible may assist in accuracy, completeness and overall survey design understanding.

- Using SDL within SDLTool requires knowledge of the abstractions and languages but tool support to advise users of mistakes or to suggest improvements to constructs and approaches is currently limited.
- Process diagrams are documentation-oriented rather than executable; supporting the latter would be helpful to assist in further automation of survey implementation.
- While the web services generated are reusable, only limited off the shelf technique implementation tools are currently supported.
- Limited and fairly generic support for version control, reusable patterns for various diagram types, diffing and merging, check-in/checkout etc are provided via Pounamu. While these are quite powerful, tailoring many of them more to SDLTool user's needs would make the tool both easier to use and more effective.

Overall, we judge that use of SDL and its supporting environment, SDLTool support an affirmative answer to RQ1 (Can visual notations with appropriate software tool support be used by end users, i.e. statisticians and other survey developers, to facilitate statistical survey design and implementation?).

- We have been able to provide designers with explicit, domain-specific visual language (DSVL) representations of the overall statistical survey process, many common tasks, and statistical survey data and techniques. Novel concepts such as our survey diagrams and task diagrams could usefully be adopted to describe complex statistical surveys with or without using SDLTool itself.
- Using the DSVLs that we developed for SDL does appear to provide both an effective and efficient approach to survey design. It is unclear how much benefit this is to expert designers, though our experts were positive about the more tangible representations of aspects of survey design in SDL.
- For novice and intermediate survey designers, SDL and SDLTool provide enhancements over existing batch-oriented, low-level tools. Our web service generation support is relatively novel compared to most existing platforms. Experts are generally comfortable with using existing tools but see benefits for documenting surveys that lack suitable current notations, and for teaching statistical survey design.

We have adopted a range of approaches to the development and evaluation of SDL and SDLTool that we hope other DSVL developers will find useful. We briefly summarise the key approaches we adopted and lessons we have learned from their application in this research project.

- Multiple techniques for requirements gathering and validation: as we were not experts in the target domain of this research, we found that we had to adopt a range of information gathering approaches, and multiple ways of validating the information gathered. As described in Section 4, this included corpus analysis using natural language extraction; user-centric design, visual language design principles and noticing the similarity of statistical survey design process structures to software process structures. On this project we found all of these provided useful guidance. On other DSVL projects, we may have more expertise or may find other techniques helpful e.g. more formal comparative analysis to existing approaches, brainstorming solutions with focus groups etc.
- Principles for DSVL notational and meta-model design: this work was carried out guided by an early CD evaluation of an early SDL design, using Burnett et al's visual language design guidelines, using general usability principles, and was constrained by our meta-tool platform, Pounamu's, capabilities, in terms of visual language rendering and editing support. The latter, one might argue, should not overly constrain DSVL design, but in our experience if one wants good tool support for the DSVL, one must work within the limitations of the eventual implementation platform capabilities. Were we design SDL now, we would also inform SDL design using Moody's Physics of Notations framework [32], which we have successfully used on several of our more recent DSVL tool research projects.
- We used evaluations often and early not only in this work but many other of our DSVL tool development projects. We also used a range of evaluations – end user, heuristic (in this case, based on the CD framework), and walkthrough. We would have used comparative analysis of SDLTool to existing survey design tools – however existing tools lack a great many of SDL and SDLTool's capabilities (e.g. survey overview, task model, process model and even high-level data and technique representations) making such comparisons trite.

9.5 *Future work*

Several areas of future work have already been identified, specifically better representation of inter-process dependencies, design critics to catch errors such as the attribute attachment error [1], and multiple view/elision support for large diagrams [27]. In addition, we see considerable scope for providing back end integration with other statistical survey tools so that our SDL environment can be used to not only design a statistical survey, but also implement and control it. Pounamu has an increasingly sophisticated set of integration mechanisms, including RMI and web services based APIs, code generation and import, together with collaborative work support that allows multiple designers to use Pounamu generated tools collaboratively. These can be leveraged to integrate Pounamu with other statistical packages. However, we also see the opportunity to provide a more generic framework for integration using a meta-data based approach for specifying legacy tool capabilities. We are exploring this in current work. We plan to add executable

statistical process diagrams, user scripting via the R language for more powerful analysis capabilities for expert users, and repositories for both statistical data and reusable process models. Further enhancements include secondary notation support for annotative diagrams, visual presentation of underlying statistical survey process enactment and data analysis state, improved visual presentation of mapping forms, and automatic layout of some diagrams.

10. Summary

We have described SDL, a set of visual notations for specifying statistical surveys, and its support environment, SDLTool, with modelling support, generation and execution support for complex statistical survey implementations, and generation of reusable web services encapsulating survey techniques. SDL aims to provide a similar modelling framework for statistical survey design as UML does for software design. SDL is comprised of five diagrammatic types, two of which has multiple levels of abstraction. These together represent both the data and process-oriented components of a statistical survey design. SDLTool supports modelling with SDL and integration of rich statistical meta-data, data sets, 3rd party technique implementations for analysis, and 3rd party visualisation tools. Statistical survey technique implementations are made available to 3rd party client applications via generated web services, which are then deployed and advertised for discovery and invocation. These web services are synthesized from the domain-specific visual language models in SDLTool. A cognitive walkthrough with a subject expert was undertaken to evaluate the usability of SDL. In addition, we undertook a larger cognitive dimensions informed usability study of SDLTool with eight undergraduate statistics students. We have also generated web service interfaces to several complex statistical survey techniques defined and tested in SDLTool. We have used these web services from remote clients to send data to the technique's web service, have this processed by the 3rd party analysis tools used to realise the technique's implementation, and return data to the client for further analysis and result visualisation. Service and statistical survey documentation is also generated by SDLTool to aid understanding and reuse. Combined these evaluations provided support for the efficacy of our approach.

Acknowledgements

The authors gratefully acknowledge the assistance of James Reilly as statistical consultant, our panel of user evaluators, and Nianping Zhu for implementing the Pounamu meta tool. Support from the Foundation for Research, Science and Technology for parts of this research is gratefully acknowledged.

References

- [1] Ali, N., Hosking, J.G., Huh, J. and Grundy, J.C. Template-based Critic Authoring for Domain-Specific Visual Language Tools, *2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, Cornwallis, Oregon, USA, Sept 20-24 2009.
- [2] Biemer, P.P. and Lyberg, L.E. *Introduction to survey quality*, Wiley Inter-Science 2003, Chapter 2.
- [3] Biezunski, M Topic Maps at a glance, *XML EUROPE '99*, Granada, Spain, 26-30 April 1999.
- [4] Blackwell, A.F., "First steps in programming: a rationale for attention investment models," *IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Arlington, Virginia, USA, September 3 – 6, 2002.
- [5] Bottoni, P., Costabile M. F., Levialdi, S. , Matera, M., Mussio, P., Principled Design of Visual Languages for Interaction, *IEEE Symposium on Visual Languages*, Seattle, Washington, USA, Sept 10-14 2000, pp. 45-52.
- [6] Burnett, M., Baker, M., Bohus, C., Carlson, P., Yang, S., van Zee, P., Scaling Up Visual Programming Languages *IEEE Computer*, March 1995, 45-54.
- [7] Chambers, J.M. and Ryan, B.F., The ASA Statistical Computing Section: A History, *The American Statistician*, 4 (2), May 1990, pp 87-89
- [8] CSPro, <http://www.census.gov/ipc/www/cspro/>
- [9] DDI, The Data Documentation Initiative, <http://www.icpsr.umich.edu/DDI/>
- [10] Danado, J., & Paternò, F. Puzzle: a visual-based environment for end user development in touch-based mobile phones. *Human-Centered Software Engineering*, Springer Berlin Heidelberg, 2012.
- [11] Daniel, F., Soi, S., Tranquillini, S., Casati, F., Heng, C., & Yan, L. From people to services to UI: distributed orchestration of user interfaces. *Business Process Management*, Springer 2010, pp. 310-326.
- [12] Doderò, J. M., del Val, Á. M., & Torres, J. An extensible approach to visually editing adaptive learning activities and designs based on services, *Journal of Visual Languages & Computing*, 21(6), 2010, 332-346.
- [13] Fogli, D., & Provenza, L. P. End-user development of e-government services through meta-modeling. In *End-User Development*, Springer, 2011, pp. 107-122.
- [14] Gillman, D. and Appel, A. The Statistical Metadata Repository: an electronic catalog of survey descriptions at the U.S. census bureau, *IASSIST Quarterly*, Summer 1997.
- [15] Garg, P. and Jazayeri, M. Process-Centred Software Engineering Environments, *Software Process*, Wiley, pp25-49, 1996
- [16] Green, T.R.G and Petre, M, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing*, 7 (2), June 1996, pp.131-174.

- [17] Green, T. R. G., Burnett, M. M., A Ko, J., Rothermel, K. J., Cook, C. R., and Schonfeld, J., Using the Cognitive Walkthrough to Improve the Design of a Visual Programming Experiment *IEEE Symposium on Visual Languages*, Seattle, Washington, Sept. 2000, 172-179.
- [18] Ihaka, R. and Gentleman, R. R: A Language for Data Analysis and Graphics, *Computational and Graphical Statistics*, 5 (3), 1996, pp. 299-314.
- [19] Jenkins, S. G. The Triple-S survey interchange standard <http://www.triple-s.org/sssasc96.htm>
- [20] Kugler, H., Larjo, A., & Harel, D. (2010). Biocharts: a visual formalism for complex biological systems, *Journal of The Royal Society Interface*, 7(48), 1015-1024.
- [21] Kim, C. H. *Visual Language and Environment for Statistical Surveys*, MSc Thesis, Department of Computer Science, University of Auckland, February 2006.
- [22] Kim, C. H., Hosking, J., Grundy, J., Model Driven Design and Implementation of Statistical Surveys, *40th Annual Hawaii International Conference on System Sciences*, Hawaii, 3-6 Jan 2007.
- [23] Kim, C., Hosking, J., and Grundy, J., A Suite of Visual Languages for Statistical Survey Specification, *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, USA, 20-24 Sept 2005, 19-26.
- [24] Knuplesch, D., Reichert, M., Ly, L. T., Kumar, A., & Rinderle-Ma, S. Visual modeling of business process compliance rules with the support of multiple perspectives. *Conceptual Modeling*, Springer, 2013, pp. 106-120.
- [25] Lanzenberger, M., Sampson, J., & Rester, M. Ontology Visualization: Tools and Techniques for Visual Representation of Semi-Structured Meta-Data, *Journal of Universal Computer Science*, 16 (7), 2010, 1036-1054.
- [26] Lieberman, H., Paternò, F., and Wulf, V., *End user development*. Human-Computer Interaction Series, Vol. 9. Springer, 2006.
- [27] Li, L. Hosking, J.G. and Grundy, J.C. Visual Modelling of Complex Business Processes with Trees, Overlays and Distortion-Based Displays, *2007 IEEE Symposium on Visual Languages and Human-Centric Computing*, USA, Sept 23-27 2007, IEEE CS Press.
- [28] Liu, H. and Lieberman, H. Toward a Programmatic Semantics of Natural Language, *2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September 26-29, 2004.
- [29] Madansky A., On Biblical Censuses, *Journal of Official Statistics*, 2 (4), 1986. pp. 561-569
- [30] Mellor, S.J., Scott, K., Uhl, A., Weise, D. *MDA Distilled: Principles of Model-Driven Architecture*, Addison-Wesley, 2004.
- [31] MetaNet, 2003 *Metanet Project Meeting Final Report*, Chapter 1, Samos, May 2003, http://data-archive.ac.uk/media/1689/METANET_proceedings_finalreport.pdf
- [32] Moody, Daniel. The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering, *IEEE Transactions on Software Engineering*, 35 (6), 2009, pp. 756-779.
- [33] NZCS 2003, New Zealand National Survey of Crime Victims in 2001, New Zealand Ministry of Justice
- [34] Office for National Statistics (ONS) UK , *UK Labour Force Survey Statistical Outputs Group*, <http://www.statistics.gov.uk/STATBASE/Source.asp>
- [35] Olenski, J. Global Standard for Harmonization of Social Statistics, *Expert Group Meeting on Setting the Scope of Social Statistics United Nations Statistics Division in collaboration with the Siena Group on Social Statistics*, New York, 6-9 May 2003, https://unstats.un.org/unsd/demographic/meetings/egm/Socialstat_0503/docs/no_10.pdf.
- [36] OMG, *OMG Unified Modelling Language*, <http://www.uml.org/>
- [37] Object Management Group, *What is OMG-UML and Why is it important?* <http://www.omg.org/new/pr97/umlprimer.html>
- [38] Pederson, D., Evans, B., Chance, G., Bento, A. and Fox, A. Predictor of One-Year Development Status in Low Birth Weight Infants, *Journal of Developmental and Behavioural Paediatrics*, 9 (5), Oct 1988, p287-92.
- [39] Pepper, S., The TAO of Topic Maps, *Ontopia AS*, 2002, <http://www.ontopia.net/topicmaps/materials/tao.html> SAS Institute Inc. <http://www.sas.com>
- [40] SPSS, SPSS Inc. *SPSS statistical software*, <http://www.spss.com>
- [41] SPSS Inc., *SurveyCraft*, <http://www.spss.com/surveycraft/>
- [42] Statistics Netherlands, *Blaise*, <http://www.cbs.nl>
- [43] Sutcliffe, A. *Domain Theory: Patterns for Knowledge and Software Reuse*, Lawrence Erlbaum Associates, Inc. Mahwah, NJ, USA, 2002
- [44] Unhelkar, B., and Henderson-Sellers, B., Modelling Spaces and the UML, 2004 *Information Resources Management Association Conference*, New Orleans, 2004
- [45] Wajid, U., Namoun, A., & Mehandjiev, N. (2011). Alternative representations for end user composition of service-based systems. *End-User Development* (pp. 53-66). Springer Berlin Heidelberg.
- [46] Young, F.W. & Bann, C.M. *ViSta: A Visual Statistics System Statistical Computing Environments for Social Research*, Sage Publications, Inc., 1997, 207-235
- [47] Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, 2004 IEEE Symposium on Visual Languages and Human-Centric Computing, Rome, Italy, 25-29 Sept 2004, pp. 254-256.

7.6 Engineering Complex Data Integration and Harmonization Systems

Avazpour, I., Grundy, J.C., Zhu, L., Engineering Complex Data Integration and Harmonization Systems, *Journal of Industrial Information Integration*, vol 16, Elsevier, Dec 2019

DOI: [10.1016/j.jii.2019.08.001](https://doi.org/10.1016/j.jii.2019.08.001)

Abstract: Complex data transformation, aggregation and visualization problems are becoming increasingly common. These are needed in order to support improved business intelligence and end-user access to data. However, most such applications present very challenging software engineering problems including noisy data, diverse data formats and APIs, challenging data modeling and increasing demand for sophisticated visualization support. This paper describes a data integration, harmonization and visualization process and framework that we have been developing. We discuss our approach used to tackle complex data aggregation and harmonization problems and we demonstrate a set of information visualizations that can be developed from the harmonized data to make it usable for its target audience. We use a case study of Household Travel Survey data mapping, harmonization, aggregation and visualization to illustrate our approach. We summarize a set of lessons that we have learned from this industry-based software engineering experience. We hope these will be useful for others embarking on challenging data harmonization and integration problems. We also identify several key directions and needs for future research and practical support in this area.

My contribution: Developed many of the key research ideas, co-authored significant parts of the paper, co-investigator for funding for this project from ARC and co-investigator for funding from AURIN

Engineering Complex Data Integration, Harmonization and Visualization Systems

Iman Avazpour^a, John Grundy^b, Liming Zhu^c

^a*School of IT, Deakin University, Burwood, VIC 3125, Australia*

^b*Faculty of IT, Monash University, Clayton, VIC 3800, Australia*

^c*Software & Computational Systems, Data61, CSIRO, Australia*

^d*School of Computer Science and Engineering, University of New South Wales, Sydney, Australia*

Abstract

Complex data transformation, aggregation and visualization problems are becoming increasingly common. These are needed in order to support improved business intelligence and end-user access to data. However, most such applications present very challenging software engineering problems including noisy data, diverse data formats and APIs, challenging data modeling and increasing demand for sophisticated visualization support. This paper describes a data integration, harmonization and visualisation process and framework that we have been developing. We discuss our approach used to tackle complex data aggregation and harmonization problems and we demonstrate a set of information visualizations that can be developed from the harmonized data to make it usable for its target audience. We use a case study of Household Travel Survey data mapping, harmonization, aggregation and visualization to illustrate our approach. We summarize a set of lessons that we have learned from this industry-based software engineering experience. We hope these will be useful for others embarking on challenging data harmonization and integration problems. We also identify several key directions and needs for future research and practical support in this area.

Email addresses: iman.avazpour@deakin.edu.au (Iman Avazpour),
john.grundy@monash.edu (John Grundy), Liming.Zhu@data61.csiro.au (Liming Zhu)

1. Introduction

One of the most common problems in computing is the need to integrate multiple sources of information represented in disparate data representations in order to leverage the combined information i.e. to “harmonize” the disparate data into a single, consistent form [1, 2, 3]. Once systems are created, changing the data formats is very expensive due to engineering and change impact propagation. When integrating data sources with a diverse set of federated owners, changing them can be impossible due to ownership and even legal issues in government datasets.

Various research and industrial applications have been working on developing such data mapping and aggregation solutions in order to make transitioning from one data format to another less expensive and more user-friendly format (e.g. [4, 5, 6, 7]). This is becoming an increasingly common problem with the increase in availability of large and open datasets and demand for such data integration, ongoing updates, analysis and visualization [8], while addressing privacy and security concerns [9]. Developing an automated end-to-end process to support data wrangling and harmonisation will potentially result in a data product of limited quality [10]. On the other hand, it can be argued that without suitable visualizations, understanding and using such integrated large data in these aggregated systems quickly becomes very cumbersome [11]. Many users are not familiar with the low-level representation of data that is often targeted to specific technical audiences and applications [5]. To address this limitation, standard and familiar visualizations need to be incorporated that use the integrated data sub-systems. Key Example application domains for these approaches include healthcare – integrating Electronic Medical Records from disparate systems and providers; “smart city” transport systems – integrating traffic, road usage and control data from multiple systems; and land information systems – integrating GIS, land usage, and agricultural data.

This paper reports on our efforts to develop a software engineering process for complex data integration, harmonization and visualization problems. We describe the process and a prototype toolset, Harmonizer+, that we implemented to achieve such diverse data integration and harmonization. Our approach involves analysis of disparate source data models and then the design of a harmonized, consistently aggregated schema that can represent the diverse source datasets. Model mappings from each source schema are devised and used to generate complex code to transform source data into

the new harmonized schema format. Data is aggregated in a consistent way to allow querying and combining but respect different source data privacy and anonymity requirements. Finally, we have developed an interactive data visualization support tool for the harmonized data set.

As a concrete industrial case study using our approach, we use a new data integration and visualization example into the Australian Urban Research Infrastructure Network (AURIN). AURIN is an Australian initiative to make a wide range of demographic, economic, social, cultural and geographic datasets available to geographers, sociologists, government agencies, businesses and ultimately citizens for multi-dataset querying, aggregation and visualization [12]. An example of such multi-dataset information are Household Travel Surveys (HTS). For example, planners would like to know how people currently travel to and from work or school; if there are socio-economic, demographic or other impacts on these travel choices, and emergent behaviors over changing time, demographic and economic conditions. AURIN envisions to provide a consistent data querying and web-based information visualization architecture. We used our Harmonizer+ process and toolset to harmonize, integrate and visualize several State government HTS datasets into the AURIN framework.

The rest of this paper is organized as follows: in Section 2 we outline our motivation and problem statement. Section 7 briefly outlines key related work. We describe our approach to realizing a consistent, integrated HTS data schema, aggregated data and data visualization support in section 3. In the following section we describe our harmonized data model development and modeling of source data to target harmonized schema transformation. Section 5 provides details of our Harmonizer+ architecture and implementation. Section 6 provides a summary of strengths and weaknesses of our approach and key lessons learned from its industrial case study application . It lists some areas for future research and practice.

2. Motivation and problem statement

The Australian Urban Research Infrastructure Network (AURIN) is a national institute aiming to gather data from participating Australian states. It provides a framework for researchers to access, investigate and use a wide range of data from across Australia [12]. Data includes census results (e.g. demographic and socio-economic profiles), geographic data (e.g. location of roads, rail, and other infrastructure), and organizational data (e.g. Com-

monwealth, State, Local organizational structures, businesses, and hospitals) among others.

Household Travel Surveys (HTS) are an example dataset that provide insights into mobility patterns and utilization of public and private transport. Across Australian states, a number of diverse HTS have been conducted by different government agencies to find out the travel behaviors of citizens. Unfortunately all states use vastly differing data formats to record survey results. Many aggregate these results using different street, locale, suburb, demographic or other categorizations. The systems supplying the data are diverse - data comes in CSV, XML and relational formats. Some systems support interactive querying while others only batch export. None provide effective visualization capabilities especially when combining the HTS data with other data.

The AURIN project wanted to integrate HTS data seamlessly into the wider project resources, including a single harmonized data model, regular data updates, multi-dataset querying, and integrated and effective visualization support. HTS data integrated with other AURIN data would enable researchers to explore and discover new knowledge around Australian's mobility patterns. It would allow planners to investigate for example, how transport infrastructure could be improved, discover relationships between travel choices, determine how travel choices are influenced, and might even allow for improvement of travel outcomes.

The above exemplar illustrates the diverse range of data sets, data integration challenges, harmonization issues and visualization needs our work is addressing. Given the diversity of data collection instruments and vastly different data aggregation methods, we need to take source datasets and integrate them into a harmonized and consistent form to be useful. Various different aggregations of the source data often need to be made available. This enables users to access data collected by different organizations and compare their information patterns. The harmonized data could also be queried with other datasets e.g. such as other demographic data available in the AURIN framework. The results of these queries could be exported as common data formats (e.g. CSV, Spreadsheets) or visualized to better enable user investigation and analysis.

The following points summarize some of the key problems that an effective data integration, harmonization and visualization solution needs to address:

- Data harmonization and integration:

- Data is sourced from a diverse range of repositories
 - Data repositories use a range of technologies to provide access
 - Data has differing levels of aggregation, often done to preserve information privacy or reduce data storage and querying costs
 - Access to disaggregated data is limited. Many systems provide batch data export to CSV, XML, Excel or relational formats rather than providing “live” access
 - Data represents different categories and classifications across data providers that need to be agreed.
 - A single, harmonized data model must be produced that is capable of representing all State HTS datasets, at disaggregated and multiple levels of orthogonal aggregation.
- Data query and visualization:
 - An interactive visualization capability is needed to allow end user access to and exploration of the harmonized data
 - Some harmonized data can be drilled down into, some cannot
 - User specific visualizations e.g. research analysts from diverse domains, government planners and ultimately business and citizen end users need to be supported

Here, we primarily focus on addressing data harmonization and integration requirements, while also describing example solutions for harmonized data query and information visualization.

3. Our Approach

Our overall approach to harmonization aims to use data transformations to make disparate available datasets *usable*, as stated by Kandel et al. [13]. The harmonized data will then be queried and a set of visualizations for the queried data will be generated to support end users (data consumers) in their analysis of the integrated data. Figure 1 outlines the approach we take and its main steps: Cleansing, Wrangling and Usage. In the following, we describe these steps used within our Harmonizer+ framework.

Data cleansing involves three main tasks: reading data documents, examining data formats provided from each state (1); identifying shared data e.g.

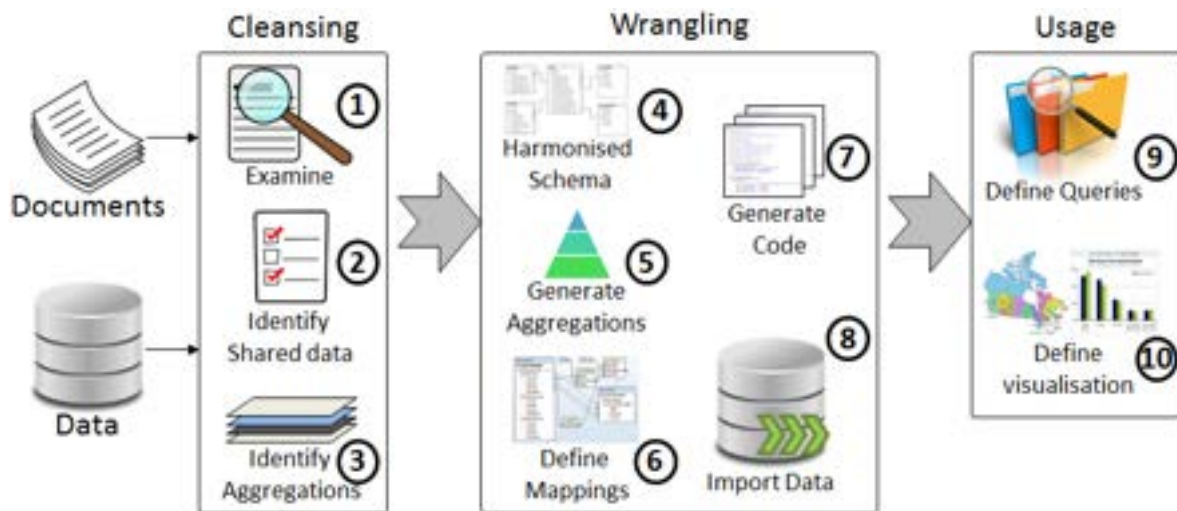


Figure 1: Outline of our approach.

locale, demographics and missing data fields (2); and identifying commonalities, differences, and aggregation levels (3). Our wrangling step includes defining harmonized schema and possible aggregation levels (4); generating aggregators for imported raw data to match other existing data (5); defining mappings from each source dataset to harmonized dataset (6); generating code based on the mapping and transformation specifications (7); and applying the mappings and transformation while importing data from various datasets (8). The final stage of our approach defines how the harmonized data is going to be used and it includes querying the harmonized data and other available datasets (9); and developing visualizations (10). In the following, we describe each step in more detail and in the context of our HTS data harmonization case study.

4. Case Study: Household Travel Survey Data Integration and Harmonization

4.1. HTS Integration Process

The problem of taking data in its initial form and transforming it into a desired form is known as data integration [14]. Success and failure of data integration frameworks is dependent on understanding the data’s context-sensitive meaning and the quality of the data [15]. Kendal et al. provide an example where a data analyst is faced with datasets provided by three states and has to perform much trial and error to cleanse the data before being able to use it [13]. Each source data set needs to be carefully understood from

available documents, schemas and example data (1). In our experience this can be challenging due to the variable quality and availability of information and often needs experimentation and domain knowledge. In the context of HTS data integration and harmonization, we first spent considerable time to understand each state’s HTS data via their documents, schemas and example data (step (1) in Figure 1). Unfortunately much of the documentation and schemas we had to work with on this project were incomplete or inaccurate. We then used a variety of tools to aid us in exploring the data and schemas including SQL Server, Altova MapForce, and Excel. This contributed to the bulk of our time (almost 60%). However, it paved the way for the rest of the harmonization to produce an effective result.

We then identified shared information, even if in different formats, between HTS schema and datasets e.g. geographic location, local government organizational units, and demographic information (step (2) in Figure 1). We identified commonalities and differences in the data item formats used to represent the same information. For example, a data field representing “Tram” as a mode of transport in one dataset may be represented as “Light Rail”, or via a nominal value (e.g. 20) in another. Differences in record structure, differences in foreign keys, differences in hierarchical structure, and differences in aggregation levels across records and hierarchies (3). For example, we were given access to raw HTS data by one state and data aggregated to household level by another state, and data aggregated to local government area (i.e. county in USA context) by another. Such issues are becoming very common; for example due to varying privacy policies across states.

Using steps (2) and (3) and end user inputs, we then designed an integrated data model broad enough to represent all data structures and fields in the source HTS datasets and their schema (4). As we are designing for a very wide range of potential end users with a goal of preserving the original data as much as possible, end user inputs did not play a major role in designing the integrated data model for this project. We then generated the aggregation procedures required to transform each HTS source data to the same aggregation level required by the harmonized schema (5). We tried using several (semi-)automated data aggregation tools, but none could meaningfully automate the aggregation in our case. We were forced to use SQL queries to manually aggregate the datasets.

The mappings between each data model’s corresponding elements and groups, and the harmonized model were defined next (6). We had to design in house methods to keep record of and document these mapping specifica-

tions. We then generate implementations of data import, aggregation, and mapping specifications (7). This can ideally be done using automated code generation facilities provided by tools like Altova MapForce. These code generation facilities usually take source and target schema, and specifications of import, aggregation and mappings and automatically generate the required code in multiple programming language syntax. This is particularly useful if the integration project is to be embedded inside another framework. The generated code needs to then run on each source system dataset and be carefully checked for errors, correcting mappings and regenerating code where necessary (8). Data import needs to run every time new data becomes available and depending on source of the data, can be batch execution or applied on stream data. Any major changes in new data formats may trigger the repeat of early tasks. This was a highly iterative process.

Final stage of the harmonization and integration is to define queries over harmonized data and other externally-stored data (9). The data was then used to design and generate interactive visualizations with CONVERt [16, 17] data mapping and visualization toolset (10). Through following sub-sections we will describe more details on the approach and its usage on the case study example of HTS data harmonization.

4.2. Harmonized HTS Data Model

Household Travel Surveys (HTS) are conducted to provide record of how people travel in a specific geographic area. These surveys are generally designed to investigate “how people utilize public transport or personal vehicles”, “what are the main purposes of their travel”, and some “demographics” for example what is the structure of households (household size, occupations, etc.).

At the starting point of this project, we had received four datasets from three Australian states namely New South Wales (NSW), Victoria and Western Australia. NSW had provided two sets of data with different aggregation levels. Ideally, there would be one single, consistent, agreed and detailed HTS data model that all Australian states use - or even an international standard allowing cross-country comparison. Unfortunately, such a model does not exist. In practice the states all use vastly different models, designed for different purposes, different data collection strategies and tools, different databases, and different underlying data formats. This will be a recipe for many inconsistencies across the datasets. In the following subsection, we

provide details of the inconsistencies we encountered and how we addressed them.

4.2.1. Major Inconsistencies Encountered

Given that the travel surveys were conducted using different survey instruments and by separate organizations, we were faced with many inconsistencies in the data. We have grouped these inconsistencies into five categories described below.

Different types of data access points. The data access points each state provided were very different. Some states provided web service and/or database access, others batch query results in form of XML or CSV file dumps.

Different high level data structure. With different data access points, data samples also came with different high level data structure. For example one sample used a relational database with various tables, while another sample was one CSV file that could be represented as a table in any type of database system.

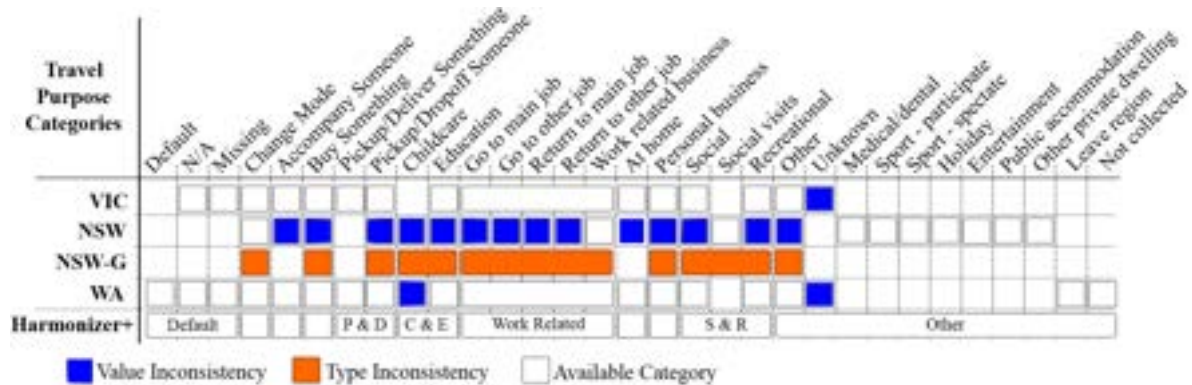
Low level data item formats. Many states had different low-level data item formats that would have to be transformed to a common representation e.g. times, dates, addresses, locations, transport modes, and purpose fields. Many numbered fields had different types as well. For example, trip distances were recorded as float, long, double, or in case of CSV files as strings.

Different coding and categorization structures. How categories are recorded were also different. For example, modes of transport could be recorded as nominal values (i.e. numbers represent modes, e.g. 2 = vehicle, 4 = public transport), as text (e.g. “vehicle”), or in separate columns (e.g. a column representing how much of distance is traveled by public transport, a column for distance by vehicle, and another distance by bike, and so on).

Missing data types, categories, or information. Since the surveys were conducted in isolation, our datasets represented many fields that were missing. This could be due to unavailability of a certain facility, lack of importance of recording an item, or different routines and procedures. For example, one state did not record how many bikes are available in each household. In another example, a state did not have Tram as a means for public transport so it was not included in the list of travel modes.

Figure 2 shows two examples highlighting inconsistencies in the categories used for *travel purpose* (e.g. work, school, leisure, health treatment) and *travel mode* (e.g. Bus, Car, Walking) across different states. Each column in the table reflects a category. Top row of the tables list all categories that

are used across all datasets. The middle four rows list different state provided datasets. Since the state of New South Wales had provided a sample aggregated data and a sample of their raw data, there are two rows associated to this state's categories (NSW and NSW-G). The bottom row lists the categories we used for our final harmonized data model.



(a) Categories used for travel purpose.



(b) Categories used for travel mode.

Figure 2: Samples of inconsistencies encountered within categories used in the datasets.

The tables of Figure 2 represent available categories by boxes. For example, the dataset provided by state of Victoria includes a travel purpose for *Accompanying Someone*, as a result a box is put in the corresponding **VIC** row and *Accompany Someone* column. Same is true for **NSW** and **WA**. Where the category is missing in the dataset, the representative cell in the table is missing. For example **NSW-G** does not provide information for *Accompany Someone* and therefore does not have the box. Tables of Figure 2 use colors to reflect different types of inconsistencies discovered in these datasets. Low level inconsistencies are shown by *Orange*, Categorical and coding inconsistencies are represented by *Blue*. For example, categories of **NSW-G** are represented by strings while others use integer nominal values.

As a result, categories of **NSW-G** are represented by Orange. Since most of the datasets provided nominal values, there is also the possibility of different values representing different or similar categories. For example VIC and WA datasets represent *Accompany Someone* by 2 while NSW represents it by 20, as a result the box representing *Accompany Someone* in **NSW** is blue. These inconsistencies should all be considered in the design of data mappings.

Similarly, there are accumulated categories (we call them multi category). For example NSW has multiple categories indicating different types of work related purposes while VIC, NSW and WA consider an accumulative field for all work related categories. These accumulative fields can be spotted with the spanning boxes across multiple columns. We had to develop such detailed data analysis and comparison tables to aid us in determining a suitable harmonized data model that could represent all of the combined data in a single manner.

4.2.2. Data Aggregation

In addition to the encountered data inconsistencies, we were faced with data at quite different aggregation levels in the source datasets. For example, we had access to the data collected by surveys, as well as data aggregated into geographical areas, for the state of New South Wales (NSW), who could provide two sets of data. One is raw data collected from user surveys that has strict privacy requirements and cannot be released. The other is aggregated to Local Government Area (LGA) as defined by the state government. Some states provided raw data only. Other states provided aggregated data to differing levels e.g. locale, street, street groups, LGA, Statistical Area (SA2), or combinations.

For this aggregation process, AURIN partners agreed that the preferred level of aggregation be at SA2 level. However, some providers do not provide data down to this level and hence we had to compromise to the LGA level. We chose to have two sets of data: one based on available raw data and/or SA2 level (when available), and another based on aggregation of the raw data to LGA level. This would give AURIN users the opportunity to see the aggregated data first and compare it. They would then investigate the raw data below this level to drill down to the more detailed data based on differing individual surveys. In addition, unharmonized data can still be exposed directly through state-specific original datasets. Thus our compromise in using LGA vs. SA2 or removing information in our harmonized data sets, are not very detrimental or irreversible since the source data is still accessible

if needed.

We used SQL querying and Microsoft SQL Server (MS-SQL) to develop our aggregations. This decision was due to availability of information about the data and the databases structure. A temporary database was defined in MS-SQL and the raw data was imported into this temporary database. Then the required queries were defined to calculate aggregations and save as new datasets. For example, a query was written to select all travels grouped by their specific LGAs and calculate sum of the distance traveled by vehicle. Differing queries were applied to raw data and aggregates below the LGA level. LGAs were then normalized to a consistent set across Australia. The process was performed once and the required queries for load, transform and export were registered as batch operations for future reuse.

4.2.3. Harmonized Data Model Design

We needed to design a new, harmonized data model for our canonical and intermediate database. This database was to be used for storing harmonized dataset where data queries for visualization would be executed on. Additionally, it would play a role in bringing together the different available datasets providing a unified data schema to map the original data to.

For the design of this harmonized data model, we needed to consider some important data limitations. Since the provided multi-state data had a variety of inconsistencies, we had three options for designing the data model: accepted majority; available in depth; or a combination of both. Accepted majority indicates the data that is available in most provided datasets. This would have helped to simplify implementation of the required data mappings. However, it would also mean that we had to remove some information provided by different states. Available in depth on the other hand, would allow us to keep all provided information, but at the expense of some incomplete datasets. Although available in depth information would not reduce the information, it would not provide a dependable platform for comparison of the data provided by different states. Additionally it would pose problems for visualization frameworks as they then need to cope with missing data. We chose the third approach which is a combination of both.

To determine how to best organize the harmonized dataset, we interviewed a sample of potential end users for the Harmonizer+ platform to see what their needs would be for this harmonized dataset and its usage. We interviewed three social science researchers who were among the end users of the final system and asked them a range of questions about their needs.

This included: what they would like to have/see in the data? what data aggregations and data elements they particularly need in their research? what platforms and data they are currently using? and what they would find useful features to have in the harmonized dataset? We also asked them about their information querying and visualization needs. Some key findings of these interviews regarding *data availability* and *data usage* are summarized below.

Data availability issues:

- Higher release frequency of data from per year to per quarter or half year - the researchers need updated HTS data more frequently than a per-year release. Some surveys are conducted quarterly or even monthly. Our Harmonizer+ solution must support update of the HTS datasets at least quarterly.
- Future-proofed support for other units of data collection beyond household - researchers see the need for data collected from other societal groups in the future, such as by organization or by venue. These need to be provided by design in our solution even though not currently available, to avoid major re-engineering efforts later.
- Targeted in-depth profiling of areas (such as disadvantaged areas and individuals) - as social scientists, these end users want to tag data, where possible, with demographic, socio-economic and other information, but with privacy considerations in mind i.e. individual or family privacy must not be breached.
- More context information about the data, including historical data - researchers want to be able to query and visualize information relating to e.g. long term changes in travel behavior due to suburb development, infrastructure development, changes in demographics etc. This means we must keep more contextual information with HTS data items as well as historic survey data and be able to link it with newer survey data.
- Common data format and data model from different states - a key goal of the project. However, additional states must be able to be added that themselves come with different HTS datasets, thus forward-looking design is essential to enable incorporation of foreseeable data not in the current models.

- Integrating with per-state data privacy policies - states have differing data privacy legislation and care must be taken not to breach any or collectively when making aggregated data available to Australia-wide researchers.
- Configurable data sets to allow different types of access from different types of users such as free or paid subscribers, privacy-compliance agreement signed or not signed. For example, some data attributes or operations on attributes may be removed for different user groups.
- Ability to accommodate continuous surveys in future. For example, user mobile devices that automatically track travel patterns and users supply additional information on purpose of travel and travel mode choice.

Data Usage issues:

- Highly customizable reports including visual reports - researchers need much more customizable reports than the existing AURIN system which incorporate primarily visual abstractions. End user specification of reports would be ideal.
- Visual graph for representing data set relationships - including support for recommending possible combinations and extensions of one data set to another e.g. suggesting combining specific locale, socio-economic or other filter/group-by.
- Mechanisms for easy linking to other attributes available in other data sets - this includes information such as income, house price, school support, access to services and goods transportation. Ideally, we would allow these users to arbitrarily integrate additional information.
- “Playground functionality” - this includes supporting what-if questions around how a data change would affect some other related attributes without deep domain knowledge. The ARUIN portal provides some basic statistical tools such as regression or descriptive statistics to allow explorative studies. The playground would allow users to include/exclude certain attributes or adjust weightings for regression analysis or descriptive results. For example, a HTS data user may want to try to do a quick examination of an attribute in another data

set (e.g. house price, walkability metrics of a suburb) to see if it has impact on total trips, before deciding to download that data set.

- Support for widely differing user group needs - this includes social science research experts, citizens, journalists, rare and daily users.
- Live on-line support for data end users - this includes pre-packaged and reusable queries and visual reporting.
- Better support of zoom-in functionalities in visualization - The default AURIN visualizations provide aggregated visualizations in charts and heat maps. These visualization, although necessary for abstract insights, does not provide detailed information of the surveys.

Based on our analysis of the base data sets commonalities and differences, and the findings from our end user interviews, we chose the following features and information to keep and to omit respectively from the datasets. Our approach was to define two separate data models and intermediate databases, one for raw data and one for aggregated data. Parts of our intermediate aggregate and harmonized database are shown in Figure 3. The main tables are HTS_Data and Travel. HTS_Data provides basic information about the surveyed LGAs including total households, average people per each household and total vehicles. Travel table provides more fine grained information regarding types of trips, time, distance, purpose and mode. The values in these tables are estimated according to surveyed participants responses.

Tables Travel, HTS_Data and Population are essentially linked by a one to one relationship, which is not a usual database design practice. We made the decision to separate them for the following reasons: HTS_Data in its current form provides generic information regarding LGAs that satisfies most researches needs. Travel table would provide more in depth information for interested researchers. This separation would allow faster processing for generic purpose querying in our on-line architecture. Only one state records participants' age as age groups. As a result, this information is recorded in separate tables. Similar to Population table, information regarding Vehicles including types, fuel, make and models, Work including occupations and income were not provided by states specially due to privacy concerns. We have included them in the design for future proofing the model.

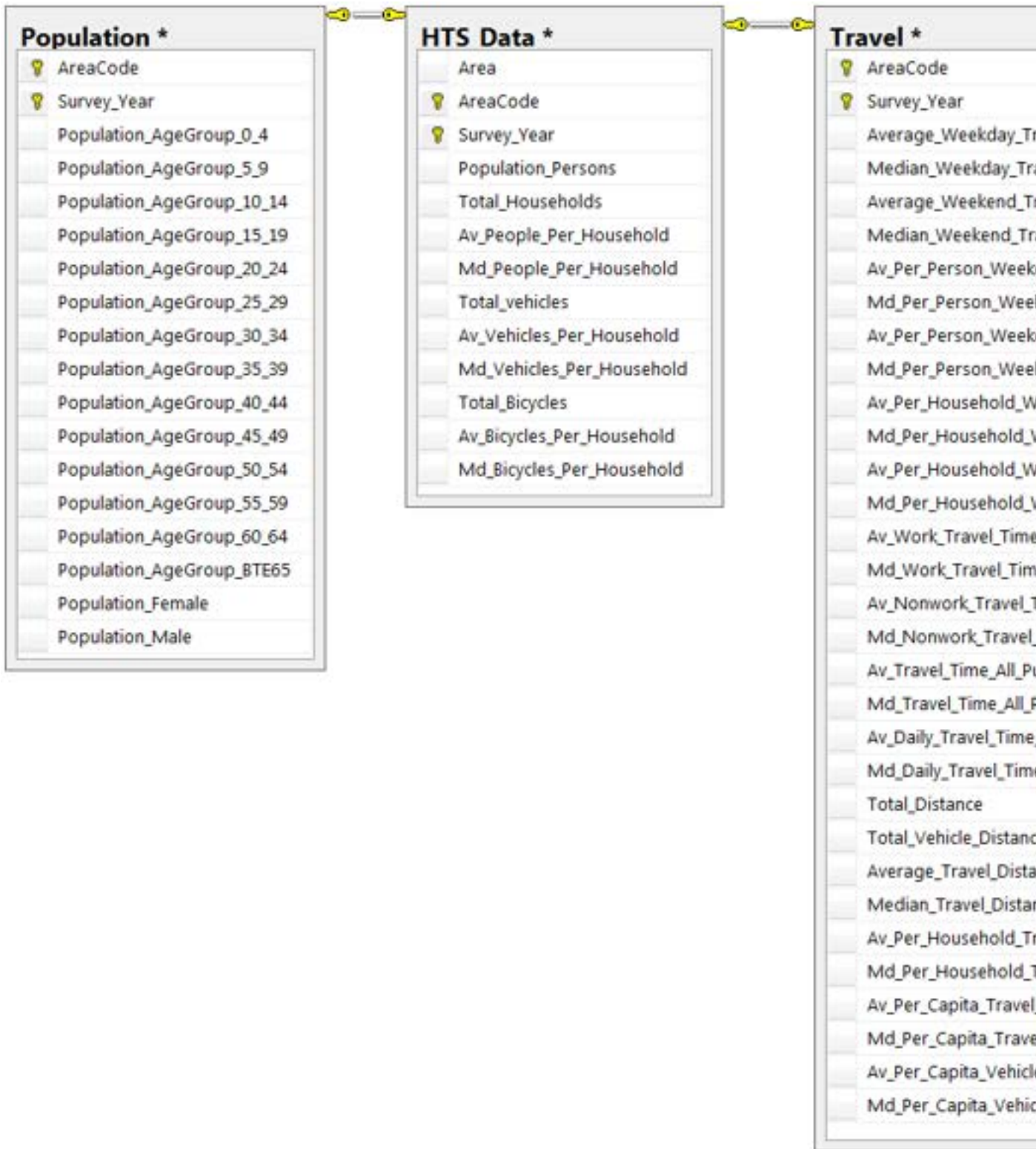
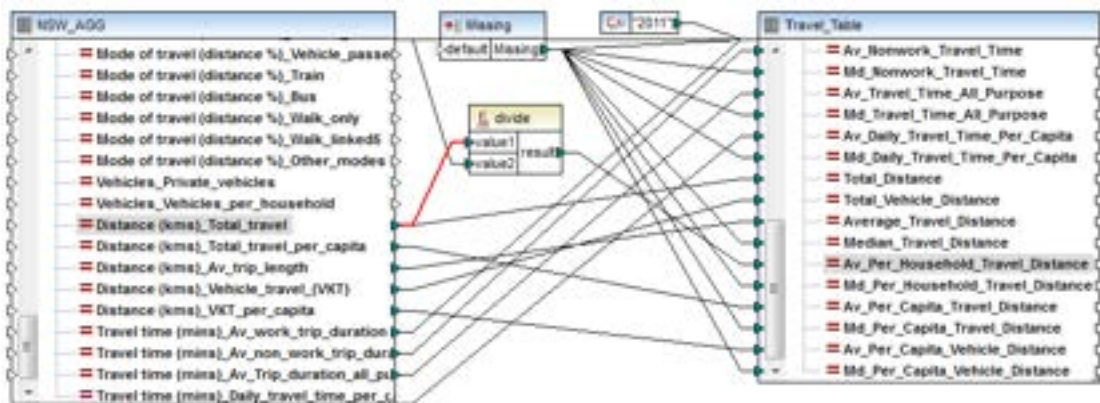
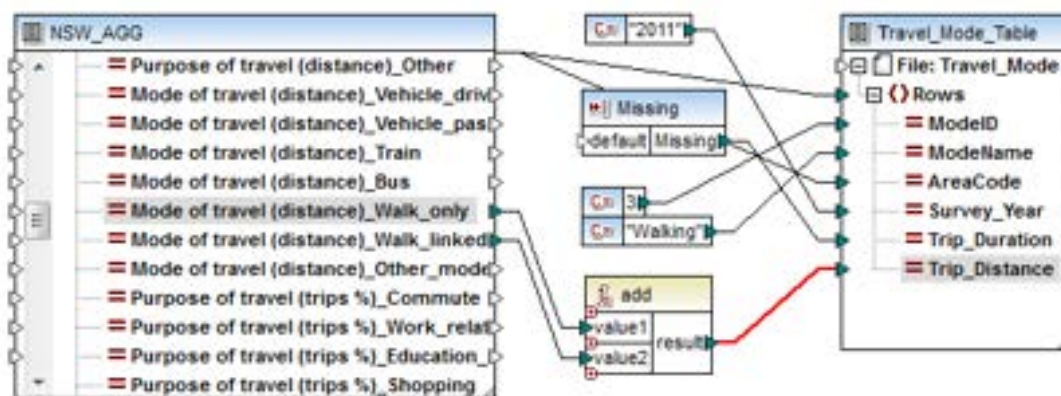


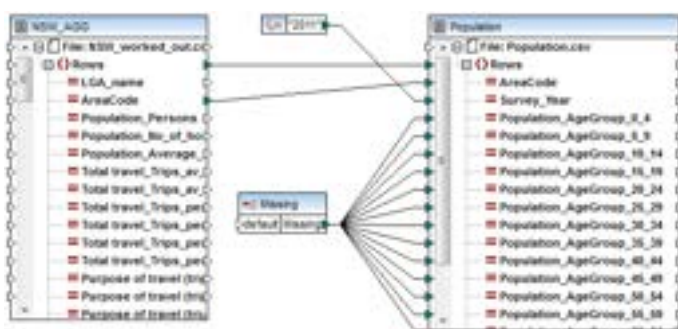
Figure 3: Part of the harmonized data model showing base travel data entities.



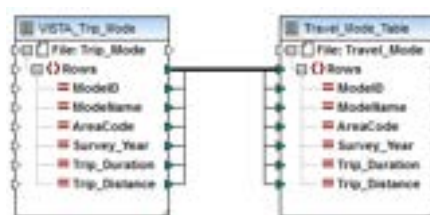
(a) Using a divide function to calculate average travel distance per household for an LGA



(b) Using add function to combined two walking related values of NSW-G dataset



(c) Providing defaults missing values



(d) Mapping calculated aggregated values.

Figure 4: Sample of mapping datasets to portion of the harmonized data model using Altova MapForce.

4.3. Defining Schema Mappings with Altova MapForce

With our harmonized data model design completed, our next step was to define the data mappings. These mappings would import the collected

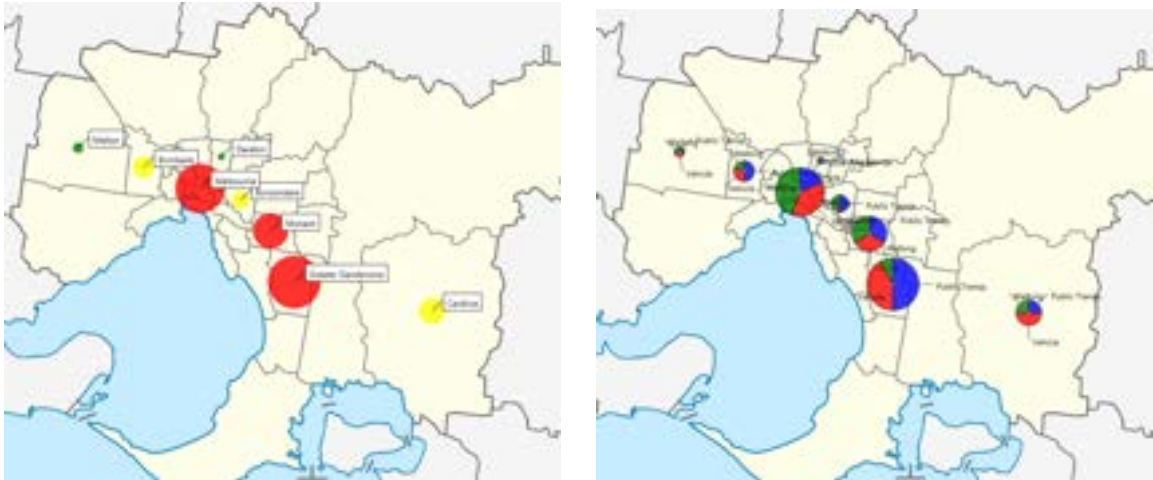
data to the new data model. We used Altova Mapforce for this data mapping task. Mapforce provides a powerful, flexible and relatively user friendly framework for complex data mapping. It provides the necessary data connectors to connect to various data sources that eliminates separate coding of the connectors. MapForce automatically generates schemas for imported data and allows viewing source and target schemas side by side. Mapping correspondences can be defined by drag and dropping elements of source and target schemas. Complex 1 to many, many to one and many to many transformations can be specified. Some such mappings we used are shown in Figure 4. From these mappings, skeleton codes for mapping and transformation implementations can be generated that are included in our Harmonizer+ framework.

The mappings depicted in Figure 4 provide various examples of fixing inconsistencies or calculating required values. For example, a divide function is used to calculate average travel distance per household for an LGA, a value that was not provided with the dataset (Figure 4(a)). Or an add function is used to combined two walking related values of NSW-G dataset (Figure 4(b)). The figure also depicts how default missing values are provided to population table where the necessary data is not available by the dataset (Figure 4(c)).

When generating aggregations of provided raw data, we had our data model in mind. That is, the raw values were aggregated to LGA level, and structured to the harmonized data model at the same time. This would provide easier integration of aggregated datasets and the harmonized data model. For example, Figure 4(d) shows how aggregated trip mode values of the dataset provided by state of Victoria is mapped to harmonized travel mode table.

From the MapForce specifications we generated a set of Java programs that extract data from each state's provided HTS dataset and import it into a single, integrated SQL Server database based on our harmonized HTS data model. The data extraction can be carried out in bulk e.g. a complete reload; or it can be done incrementally e.g. query for updated data (if supported by source system) or data in a specified area e.g. LGA (again, if supported).

We tested each data extraction program extensively to ensure consistent, harmonized data was resulting. This was a partly manual process. We had to select parts of the harmonized dataset using SQL Server queries. Then compare query results to source datasets to check for correct data extraction, data format transformation and entity/attribute mappings. During



(a) Bubble map showing total trips for a selection of Melbourne suburbs. (b) Distribution of primary mode of transport, for a selection of Melbourne suburbs.



(c) Total trips using public transport, vehicles and walking.

Figure 5: Example of Harmonizer+ visualizations.

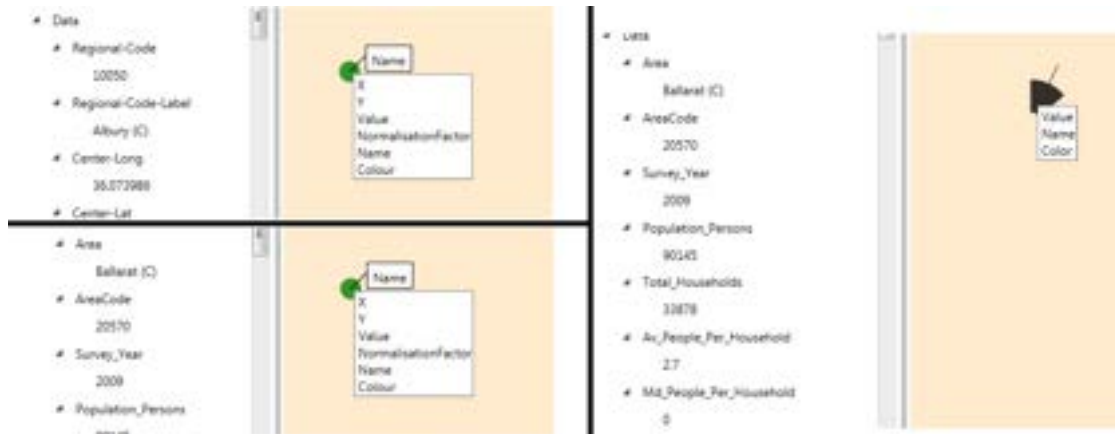
the conversion of the final data, we also did some domain-knowledge and anomaly-detection based checks on the results. For example, whether the data fits into a common-sense range, whether particular data stands out from the rest. We built some JUnit tests to semi-automate these checks so that if the data mappings and thus extractor programs are modified, many consistency checks can be repeated automatically.

4.4. Data Querying and visualization

The harmonized data available in Harmonizer+ is not that useful unless powerful and easy-to-use information visualizations are provided to end users. The current AURIN framework provides a set of default visualizations including geographic highlighting, some basic charts, and heat maps. However, the extent to which data can be explored very much depends on how many dimensions of the data can be visualized. AURIN also provides facilities for querying and exporting data. Users can select range of attributes to be included using provided GUI. A query will be generated and executed on available data and its results can be exported. This way, once the harmonized data is available in the framework, harmonized HTS data can be queried and combined with existing AURIN data e.g. household and LGA demographic and income data. However, we found the existing AURIN visualization tools very limited when trying to incorporate our harmonized HTS data. For example, while we could show total trips per suburb (or LGA) using a heat map or bubble map (see for example Figure 5(a)), it does not give any information regarding what is the mode of transport used for those trips.

Accordingly, we used an existing visualisation tool, CONVERt [17], to design set of new, more powerful and expressive visualizations for HTS and associated datasets retrieved from AURIN. CONVERt provides a by-example approach to visualization, i.e. users can import their data samples to the framework, and use available (or design new) visual notations in CONVERt and map their data using a by-example drag and drop approach. The toolset allows different notations to be composed to form complex visualizations. Returning to the mode of transport example above, we can represent a selection of transport modes by a pie chart visualization with modes as pie pieces proportional to total trips (see Figure 5(b)). Users can then compose visualization of pie charts instead of just simplistic bubbles. This way, the radius of the pie chart still represents the total trips, but its internal pie pieces will represent proportions of individual trip mode categories.

Figure 6 demonstrates the required steps for an end user for generating such a visualization. Users map their data to provided visual notations (Figure 6(a)), compose the defined notations (Figure 6(b)), and the framework will generate the required codes to generate the visualizations similar to Figures 5. These visualizations can be exported as web-enabled visualization (XAML or SVG) or as PNG images.



(a) Mapping data values to visual notations.



(b) Designing a visualisation by composing visual notations.

Figure 6: Designing a visualization in the CONVERt framework.

5. Architecture and Implementation

The architecture of Harmonizer+ solution is an ensemble of multiple components as illustrated in Figure 7. We had multiple Household Transport Survey datasets provided to us (Figure 7 (1)). Since the agreed format of use in the approach was CSV, all provided data were converted to CSV before inclusion in the approach. Depending on the level of aggregation of the provided datasets, they were sent directly to our data mapping module or through a data aggregator. Where the provided data was aggregated, the

resulting aggregated dataset was then fed to the data mapping module. Our data aggregator module (Figure 7 (2)) imports the raw state HTS data into a temporary Microsoft SQL server database, applies set of data dependent SQL queries to aggregate the data, and then exports the data as CSV files.

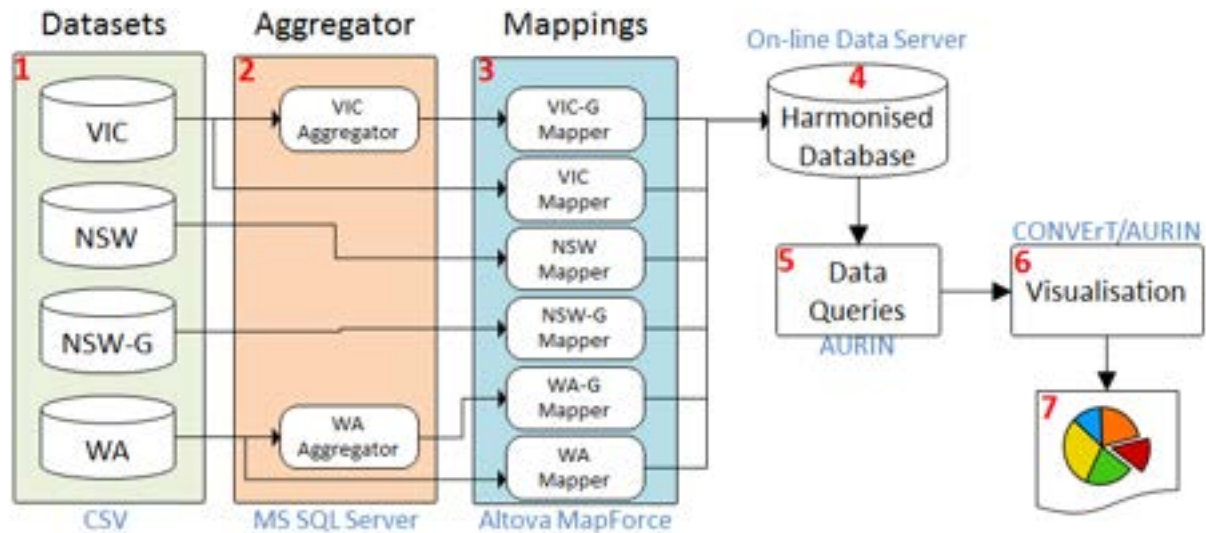


Figure 7: Data transfer and mapping procedure. Arrows indicate data flow.

Once the input data are all at same aggregation level, they are sent to our data mapping module (Figure 7 (3)). Here the set of data mappings were used to map the imported data to the harmonized data model. This module was implemented using Altova MapForce. The initial dataset specific mappings between imported datasets and intermediate schema were implemented in MapForce. The automatically generated mapping code in Java was augmented by hand to implement particularly complex transformations, and then registered as Data mapping module. The results of these data mappings are exported into the harmonized HTS database available on-line within the AURIN framework Figure 7 (4).

The AURIN framework provides facilities to query data according to its available attributes (Figure 7 (5)). Users can design and save these queries and execute them when required. The results of these queries can be used inside AURIN as spreadsheets or by set of predefined visualizations provided by the AURIN framework. Alternatively, then can be exported to CONVERt for visualization generation (Figure 7 (6)). New surveys are periodically conducted and when available their data is fed into the system. New data can be batch imported or incrementally added, depending on the capabilities to obtain it from the particular source state system.

6. Discussion and Lessons Learned

This section discusses strengths and weaknesses of our approach and user feedback on the AURIN HTS solution that we developed. We then list the lessons learned from this project and provide some pointers for future research in data harmonization approaches for complex software systems.

6.1. Strengths and weaknesses of our approach

Considering the requirements laid out in Section 2, our Harmonizer+ has met these requirements for our case study application. We have developed a forward-looking, harmonized data model able to incorporate all important aspects of the disparate current State HTS survey data. This has served as the source of a single, aggregated HTS dataset that has been incorporated into the AURIN portal. AURIN queries can be run across this integrated dataset combining with other AURIN datasets. Our CONVERt-implemented visualizations provide user-friendly, extensible visualization capabilities. However, the framework can be complex at times for generating complex visualizations.

Feedback from AURIN researchers was highly positive. All key HTS data from each state is accessible in a single, integrated form. It can be queried and visualized in highly useful and effective ways. New state data - including Queensland and Western Australia - is expected to be integrated with the existing harmonized data schema. Improved visualization using CONVERt-style by-example specification is attractive to AURIN end users.

It is challenging, even with domain experts available to the data integration team, to understand complex source system data, schema, access support (querying), and underlying technologies, especially for legacy systems. We spent a large amount of time in the AURIN HTS project trying to understand source data meaning, especially some of the complex classifications and attribute values we had to harmonize. A lack of tools to support this in the data wrangling led us to develop some support in our Harmonizer+ platform to explore source datasets and source system querying, export and API interfaces.

In terms of the completeness of the harmonized HTS data model, we had to make a trade-off between having too many missing or questionably-mapped values in a union-of-all-fields style and omitting some important data. For the most part, we were able to achieve an acceptable balance between these for AURIN end users. However, all of the individual state data is still available in its original form and original (dis-)aggregation level

if really needed i.e. via its source data portal interface. Expert users may have access to unharmonized data and the detailed mapping functions so we opted for removing some information to enable a wider-level of users, including ultimately citizens and journalists, to better understand the data and combine and query HTS data with other AURIN data.

In general, developers of integration systems will face similar data harmonization problems. Particularly challenging issues include handling source data that uses very different identifiers, very different classifications and taxonomies, different grouping and repeating structures for records, and different aggregation levels. Data can not usually be disaggregated, meaning some systems may provide data at vastly different levels of aggregation, limiting the complete set of harmonized and integrated data. Different schema designs may handle repeating records i.e. multiple instances of records or sub-records. These can be very challenging to map into a single harmonized format.

There were interesting and important privacy concerns that arose during our work. States need to ensure dis-aggregated or small locale area data does not compromise citizen privacy, and different states use different concepts of privacy. This presents a challenge when trying to harmonize the disparate source data. Removing some fields or aggregating data to highly levels to preserve privacy has an impact on research using the harmonized dataset: removing columns (attributes) may make less research possible for AURIN end users. On the other hand, removing some rows due to privacy concerns may significantly skew research results. This issue has also been noted in recent research relating to releasing MOOC data [18]. In general, data provenance, privacy and security issues are extremely important to end users, organisations and often have legislative constraint. We wanted to integrate support for data provenance and privacy management into our Harmonizer+ toolset.

6.2. Summary of lessons learned

In this section we provide a list of lessons we learned from our harmonization project and hope to draw set of future research directions in similar data harmonization cases.

Documentation: A large part of our time in the AURIN HTS project was spent on reading and understanding data documents. Where these documents were not provided, we had to reverse engineer or generate them by investigating the datasets. Often these investigations forced us to conduct

multiple question sessions with data providers. Additionally, when documents are not specifically designed for software engineers and data experts, it is very hard to understand them from the technical point of view. In one example, we were provided with a set of user manuals and data collection procedures rather than data documentation and we had to relate the provided dataset to the manuals. This proved to be a big challenge in understanding and integration of data. Additionally, once the data mappings and harmonization process was finished, we had no acceptable and agreed method of documenting our data mappings. Given the importance of understanding data mappings in similar projects, standard data mapping documentation must be available for future maintenance.

Tool support: Most of the tools we tried had very specific and limited functionalities in comparison to the full life-cycle of our data harmonization project. Most visualization tools, for example, assume data is clean and data wrangling tools mostly do not provide flexible visualizations. It is necessary to have easy and accessible to use harmonization tools. Learning the available tools to perform data aggregation and data wrangling proved a very long learning curve. As a result, our decision was to use our available expertise, and invest more time on understanding the data. Research in more user-centric approaches for performing both tasks will help the data analyst community and other harmonization projects.

Raw data: When it comes to the notion of *raw data*, different stakeholders have their own interpretations. For example, we had data provided to us in form of text (e.g. csv), processed statistical files (e.g. SPSS), or as exported databases (e.g. Access DBs). Our decision was to use the lowest level of the data, i.e. text files (csv). While transforming to lowest level is most of the times possible, it might be beneficial to use the data in higher levels specially when dealing with large databases. When collecting information, it is essential for organizations to consider as fine-grained data as possible. It is very hard to disaggregate information if not impossible. When access to fine-grained data is provided, aggregations can be generated according to the problem at hand.

Use of Models: Many areas of software engineering are benefiting from the use of model based approaches, e.g. data transformers and visualization. This can provide better testing facilities, less need for implementation in low level coding, better scalability and validation, to name a few. We hope to see more use of model-based approaches defined as round-tripping processes. This could benefit documentation i.e. use models to document the process

(e.g. data aggregation and mappings) and generate part of the final code automatically. In our example, use of automatic code generation facilities of Altova MapForce greatly improved productivity.

In our example, the automatically generated mapping code we used to transform source HTS data from states into our integrated Harmonizer+ repository proved to be highly effective. The use of Altova MapForce greatly enhanced our ability to specify complex data mappings predominantly declaratively and generate highly efficient Java programs to carry out the data transformation and integration. Maintenance effort is relatively low for these mappings and the generated mapping code as we are able to regenerate by far the majority of the translator components from MapForce.

Privacy: We have identified an interesting new research area of dynamically integrating privacy policy (for specifying which rows/columns or operations are not allowed and for what usage situation) with data access, query and analytics support. Any data filtering or removal (especially at the row level) should be communicated clearly to inform researchers using the harmonized data of the possible impact on their research results (such as correlation studies).

Visualizations: Our observations revealed that finding inconsistencies within datasets is a crucial step in data wrangling and cleansing. With large variety of datasets, it is very hard to track inconsistencies. As a result, we chose to use visualizations to help us track these inconsistencies. The visualizations of Figure 2 are samples of these visualizations using Gant chart metaphor. More research in developing such visualizations is required in conjunction with research on clustering approaches.

Future applications: We are applying the approach presented in this paper on healthcare system integration problems. Our health provider partners have diverse systems where complex patient, diagnostic, treatment and monitoring data are stored in varied formats. These need to be brought together, harmonized and visualised in much the same way as the travel survey data case study in this paper. We also plan to apply these techniques to integrating complex traffic data sourced from several systems with our roading partners, and smart home and building sensor data.

7. Related Work

Various approaches and tools have been developed and used to perform complex data mappings for data integration and mapping scenarios. For ex-

ample, a form-based mapper was introduced to help business analyst users perform data mapping using a concrete form-based metaphor [5]; Transformations to support data integration within multiple views of source and target models [19, 20]; Mapping agents to generate automated mappings between multiple source and targets [7]; And support for mapping and transformation generation using concrete visualizations [21]. However, all of these frameworks were targeted at specific domain data mapping problems, none fulfilling examples like our AURIN HTS data harmonization needs.

Clio is an early attempt by IBM to provide data transformation and mapping generator for information integration applications [6]. Clio provided declarative mappings to be specified between source and target schemas and supported mapping generation in XQuery, XSLT, SQL, and SQL/XML queries. In our project, we were provided with raw text-based data. As a result approaches that use abstractions or schema mappings were not applicable. Although we could have reverse engineered the schemas, but that would introduce multiple problems for example incompleteness of the reverse engineered schema.

Multiple approaches exist for data wrangling and cleansing with text-based datasets including Toped++ [22], Potluck [23], Karma [24], and Vegimite [25]. These approaches however do not provide all necessary mapping facilities (e.g. reshaping data layout, aggregation, and missing value manipulation) and only support a subset of the needed transformations [13]. Scripting languages like Ajax and Potter's Wheel provide data mapping and manipulation facilities [26, 27]. These however are restricted in few set of commands and introduce difficulties in programming directly with scripting languages that is not feasible for our intended end users. More recently, new frameworks have emerged to fill the gap for data integration. Examples include Talend studio¹ (specifically Talend Open Studio for Data Integration), Altova² (specifically MapFroce), and Informatica³. These frameworks provide facilities for data integration that reduce the need to engage in low level coding of various data loaders, transformations and preparation procedures. While the end results of most such frameworks for candidate projects may be the same, working with each framework, users would be exposed to different routines and data integration life cycles. Hence, we are embarking on developing a more uniform software engineering process to cater for most data integration and harmonization projects.

Given the extent to which the data processing is being used across different domains, data wrangling community is moving towards using more

modern tools that demand less technical knowhow to perform various steps of data cleaning and integration. An example of approaches that are targeted to less technical users is Trifacta Wrangler [28]. Wrangler provides a framework where users interactively manipulate data and the system infers the relevant data transformations [29]. It provides natural language description of data mappings intended for less technical users. In our project however, we were interested in batch processing of the transformations and a complete wrangling life cycle from data cleaning to visualizations, hence we chose to use

Usability of our approach was also dependant on generating understandable data visualizations. AURIN framework by default provides a set of predefined visualizations for users to query and see sample data through standard visualisations like barchart, and heat maps. However, additional visualizations cannot be defined inside the framework. We investigated integration of other visualization tools (e.g. Protovis [30], Lyra [31]). We found these tools very sensitive to uncleaned data (for example missing data fields) and could not be well integrated in the framework. Similarly, powerful visualization scripting like D3 [32], and Vega [33] require scripting knowledge and were not suitable for use in our approach.

8. Conclusions

We have described a process and supporting framework for engineering complex data integration, harmonization and visualization systems. Our approach focuses on data understanding, development of a harmonized schema, design and generation of integration mappings and mapping support, and canonical, harmonized dataset querying for end user-oriented visualizations. We have presented an industry-based project using our harmonization approach of integrating multiple household travel surveys into an intermediate canonical database. It incorporates complex multi-source data aggregation, data mapping and transformation, and information visualization. Our approach is practical for industrial usage in such domains. We have learned a number of important lessons from this experience and have identified set of key directions for future research to better-support such challenges. We are

¹www.talend.com

²www.altova.com

³www.informatica.com

applying this approach to healthcare, traffic and smart home data integration, harmonization and visualisation.

Acknowledgments

The authors would like to acknowledge the support from AURIN and Data61/CSIRO for this research.

References

- [1] Y. Zheng, Methodologies for cross-domain data fusion: An overview, *IEEE Transactions on Big Data* 1 (1) (2015) 16–34.
- [2] R. Lämmel, E. Meijer, Mappings make data processing go 'round, in: *International Conference on Generative and Transformational Techniques in Software Engineering, GTTSE'05*, Springer-Verlag, 2006, pp. 169–218.
- [3] J. Grundy, J. Hosking, R. Amor, W. Mugridge, Y. Li, Domain-specific visual languages for specifying and generating data mapping systems, *Journal of Visual Languages and Computing* 15 (34) (2004) 243 – 263.
- [4] S. Kandel, A. Paepcke, J. Hellerstein, J. Heer, Wrangler: Interactive visual specification of data transformation scripts, in: *ACM Conference on Human Factors in Computing Systems, CHI '11*, ACM, 2011, pp. 3363–3372.
- [5] Y. Li, J. Grundy, R. Amor, J. Hosking, A data mapping specification environment using a concrete business form-based metaphor, in: *IEEE Symposia on Human Centric Computing Languages and Environments*, 2002, pp. 158–166.
- [6] R. Fagin, L. M. Haas, M. Hernández, R. J. Miller, L. Popa, Y. Velegrakis, *Conceptual modeling: Foundations and applications*, Springer-Verlag, 2009, Ch. Clio: Schema Mapping Creation and Data Exchange, pp. 198–236.
- [7] S. Bossung, H. Stoeckle, J. Grundy, R. Amor, J. Hosking, Automated data mapping specification via schema heuristics and user interaction, in: *19th International Conference on Automated Software Engineering*, 2004, pp. 208–217.

- [8] G.-D. Sun, Y.-C. Wu, R.-H. Liang, S.-X. Liu, A survey of visual analytics techniques and applications: State-of-the-art research and future challenges, *Journal of Computer Science and Technology* 28 (5) (2013) 852–867.
- [9] J. P. Daries, J. Reich, J. Waldo, E. M. Young, J. Whittinghill, D. T. Seaton, A. D. Ho, I. Chuang, Privacy, anonymity, and big data in the social sciences, *Queue* 12 (7) (2014) 30:30–30:41.
- [10] M. Koehler, A. Bogatu, C. Civili, N. Konstantinou, E. Abel, A. A. Fernandes, J. Keane, L. Libkin, N. W. Paton, Data context informed data wrangling, in: *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, 2017, pp. 956–963.
- [11] I. Avazpour, Towards user-centric concrete model transformation, Ph.D. thesis, Swinburne University of Technology (2014).
- [12] R. Sinnott, G. Galang, M. Tomko, R. Stimson, Towards an e-infrastructure for urban research across australia, in: *7th IEEE International Conference on E-Science*, 2011, pp. 295–302.
- [13] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, P. Buono, Research directions in data wrangling: Visualizations and transformations for usable and credible data, *Information Visualization* 10 (4) (2011) 271–288.
- [14] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, R. D. Chamberlain, Dibs: A data integration benchmark suite, in: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ACM, 2018, pp. 25–28.
- [15] M. Janssen, E. Estevez, T. Janowski, Interoperability in big, open, and linked data—organizational maturity, capabilities, and data portfolios, *Computer* 47 (10) (2014) 44–49.
- [16] I. Avazpour, J. Grundy, L. Grunske, Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations, *Journal of Visual Languages & Computing* 28 (0) (2015) 195 – 211.

- [17] I. Avazpour, J. Grundy, CONVERt: A framework for complex model visualisation and transformation, in: IEEE Symposium on Visual Languages and Human-Centric Computing, 2012, pp. 237–238.
- [18] D. T. Seaton, Y. Bergner, I. Chuang, P. Mitros, D. E. Pritchard, Who does what in a massive open online course?, *Communications of the ACM* 57 (4) (2014) 58–65.
- [19] H. Stoeckle, J. Grundy, J. Hosking, A framework for visual notation exchange, *Journal of Visual Languages and Computing* 16 (3) (2005) 187–212.
- [20] H. Stoeckle, J. Grundy, J. Hosking, Approaches to supporting software visual notation exchange, in: IEEE Symposia on Human Centric Computing Languages and Environments, 2003, pp. 59–66.
- [21] J. Grundy, R. Mugridge, J. Hosking, P. Kendall, Generating edi message translations from visual specifications, in: Proceedings of 16th Annual International Conference on Automated Software Engineering, 2001. (ASE 2001), 2001, pp. 35–42.
- [22] C. Scaffidi, B. Myers, M. Shaw, Intelligently creating and recommending reusable reformatting rules, in: 14th International Conference on Intelligent User Interfaces, IUI '09, ACM, 2009, pp. 297–306.
- [23] D. Huynh, R. Miller, D. Karger, Potluck: Semi-ontology alignment for casual users, in: The Semantic Web, Vol. 4825 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 903–910.
- [24] R. Tuchinda, P. Szekely, C. A. Knoblock, Building mashups by example, in: 13th International Conference on Intelligent User Interfaces, IUI '08, ACM, 2008, pp. 139–148.
- [25] J. Lin, J. Wong, J. Nichols, A. Cypher, T. A. Lau, End-user programming of mashups with vegemite, in: 14th International Conference on Intelligent User Interfaces, IUI '09, ACM, 2009, pp. 97–106.
- [26] H. Galhardas, D. Florescu, D. Shasha, E. Simon, Ajax: An extensible data cleaning tool, *SIGMOD Rec.* 29 (2) (2000) 590–.

- [27] V. Raman, J. M. Hellerstein, Potter’s wheel: An interactive data cleaning system, in: 27th International Conference on Very Large Data Bases, VLDB ’01, Morgan Kaufmann, 2001, pp. 381–390.
- [28] T. W. Enterprise, Trifacta wrangler (2015) (2016).
- [29] J. M. Hellerstein, J. Heer, S. Kandel, Self-service data preparation: Research to practice., IEEE Data Eng. Bull. 41 (2) (2018) 23–34.
- [30] M. Bostock, J. Heer, Protovis: A graphical toolkit for visualization, IEEE Transactions on Visualization and Computer Graphics 15 (6) (2009) 1121–1128.
- [31] A. Satyanarayan, J. Heer, Lyra: An interactive visualization design environment, in: Computer Graphics Forum, Vol. 33, Wiley Online Library, 2014, pp. 351–360.
- [32] M. Bostock, V. Ogievetsky, J. Heer, D3: Data-driven documents, IEEE Transactions on Visualization and Computer Graphics 17 (12) (2011) 2301–2309.
- [33] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, J. Heer, Vega-lite: A grammar of interactive graphics, IEEE Transactions on Visualization and Computer Graphics 23 (1) (2017) 341–350.

8

Future Directions

8.1 Towards Human-Centric Model-Driven Software Engineering

Grundy J.C., Khalajzadeh, H., McIntosh, J., Towards Human-Centric Model-Driven Software Engineering, *15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE2020)*, 5-6 May 2020, Prague, Czech Republic, SitePress, pp. 229-238.

DOI: [10.5220/0009806002290238](https://doi.org/10.5220/0009806002290238)

Abstract: Many current software systems suffer from a lack of consideration of the human differences between end users. This includes age, gender, language, culture, emotions, personality, education, physical and mental challenges, and so on. We describe our work looking to consider these characteristics by incorporation of human centric-issues throughout the model-driven engineering process lifecycle. We propose the use of the co-creational "living lab" model to better collect human-centric issues in the software requirements. We focus on modelling these human-centric factors using domain-specific visual languages, themselves human-centric modelling artefacts. We describe work to incorporate these human-centric issues into model-driven engineering design models, and to support both code generation and run-time adaptation to different user human factors. We discuss continuous evaluation of such human-centric issues in the produced software and feedback of user reported defects to requirements and model refinement.

My contribution: Developed all of the key research ideas, wrote majority of the paper, sole investigator for funding for this project from the ARC

Towards Human-Centric Model-Driven Software Engineering

John Grundy^a, Hourieh Khalajzadeh^b and Jennifer McIntosh^c
Faculty of Information Technology, Monash University, Clayton, Victoria, Australia
{John.Grundy, Hourieh.Khalajzadeh, Jenny.McIntosh}@monash.edu

Keywords: Model-driven engineering, Human-centric software engineering, human factors

Abstract: Many current software systems suffer from a lack of consideration of the human differences between end users. This includes age, gender, language, culture, emotions, personality, education, physical and mental challenges, and so on. We describe our work looking to address these issues by incorporation of human centric-issues throughout the model-driven engineering process lifecycle. We propose the use of the co-creational "living lab" model to better collect human-centric issues in the software requirements. We focus on modelling these human centric factors using domain-specific visual languages, themselves human centric modelling artefacts. We describe work to incorporate these human-centric issues into model-driven engineering design models, and to support both code generation and run-time adaptation to different user human factors. We discuss continuous evaluation of such human-centric issues in the produced software and feedback of user reported defects to requirements and model refinement.


1 INTRODUCTION


Modern software systems are extremely complex, currently hand-crafted artefacts, which leads them to be extremely brittle and error prone in practice. We continually hear about issues with security and data breaches (due to poorly captured and implemented policies and enforcement); massive cost overruns and project slippage (due to poor estimation and badly captured software requirements); hard-to-deploy, hard-to-maintain, slow, clunky and even dangerous solutions (due to incorrect technology choice, usage or deployment); and hard-to-use software that does not meet the users' needs and causing frustration (due to poor understanding of user needs and poor design) (Curumsing et al., 2019; Prikładnicki et al., 2013; Yusop et al., 2016). This leads to huge economic cost, inefficiencies, not fit-for-purpose solutions, and dangerous and lifethreatening situations. Software is designed and built primarily to solve human needs. Many of these problems can be traced to a lack of understanding and incorporation of human-centric issues during the software engineering process (Hartzel, 2003; Miller et al., 2015; Stock et al., 2008; Wirtz et al., 2009; Smith, 1998).


2 MOTIVATING EXAMPLE

2.1 Smart Home

Consider a representative example - a "smart home" aimed at providing ageing people with technology-based support for physical and mental challenges so they are able to stay in their own home longer and feel safe and secure (Curumsing et al., 2019; Grundy et al., 2018). To develop a solution the software team must deeply understand technologies like sensors, data capture and analysis, communication with hospital systems, and software development methods and tools. However, they must also deeply understand and appreciate the human aspects of their stakeholders: ageing people, their families and friends, and clinicians/community workers. These include the *Technology Proficiency and Acceptance* of ageing people – likely to be much older than the software designers. The development of "Smart homes" technology should factor in the *Emotional* – both positive and negative – reactions to the smart home e.g. daily interaction is potentially positive but being monitored potentially negative. The *Accessibility* of the solutions for people with e.g. physical tremors, poor eyesight, wheel-chair bound, and cognitive decline. Within this, personality differences may be very important e.g. those wanting flexible dialogue compared to those needing directive dialogue with the system.

^a  <https://orcid.org/0000-0003-4928-7076>

^b  <https://orcid.org/0000-0001-9958-0102>

^c  <https://orcid.org/0000-0000-0000-0000>

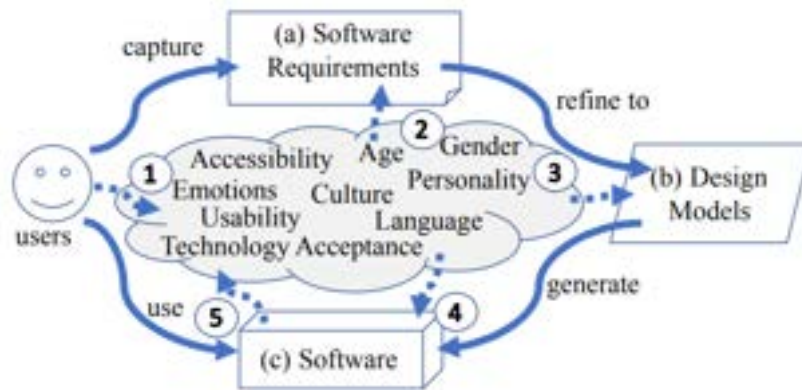


Figure 1: Incorporating “Human-centric” software issues into Model-Driven Software Engineering.

The *Usability* of the software for a group of people with varied needs e.g. incorporating the use of voice or gestures or modified smart phone interface. The ageing population is diverse and therefore smart home technology must accommodate for the different *Ages*, *Genders*, *Cultures* and *Languages* of users including appropriate use of text, colours, symbols. This is particularly important as one quarter of the elderly in Australia are non-native English speakers and the majority women, but by far the majority of software developers are 20-something year old English-speaking men. Within this, personality differences may be very important e.g. those wanting flexible dialogue compared to those needing directive dialogue with the system. Failure to incorporate human-centric issues into the development of Smart Home software has the potential to result in a home that is unsuitable for who it is designed to help, by introducing confusing, possibly unsettling and invasive, and even potentially dangerous technology.

2.2 Key Challenges

Current software engineering approaches ignore many of these human-centric issues or address them in piece-meal, ad-hoc ways (Prikladnicki et al., 2013; Curumsing et al., 2019; Hartzel, 2003; Wirtz et al., 2009; Ameller et al., 2010). For example, key aspects of Model-Driven Software Engineering (MDSE) are outlined in Figure 1. In MDSE user requirements for the software are captured and represented by a variety of abstract requirements models (a). These are then refined to detailed design models (b) to describe the software solution. These design models are then transformed by a set of generators into software code (c) to implement the target system (Schmidt, 2006). However, currently almost no human-centric issues are captured, reasoned about or used when producing or testing this software (Miller et al., 2015; Yusop

et al., 2016).

2.3 Generating More Human-centric Software

We need to fully integrate these human-centric issues into model-driven software development. In order to do this, we aim to carry out the following steps:

1. use a co-creational, agile Living Lab-based approach, enabling software teams to capture and reason about under-represented, under-used, under-supported yet critical human-centric requirements of target software;
2. develop a new set of human-centric requirements modelling languages, enabling software engineers to effectively model diverse human-centric issues, along with new approaches for obtaining and extracting such human-centric software requirements from a wide variety of sources e.g. Word, PDF, natural language, videos, sketches, ...;
3. augment conventional model-driven engineering design models with human-centric requirements for use during MDSE, along with techniques to verify the completeness, correctness and consistency of these models, and proactively check them against best practice models and principles;
4. develop new techniques to incorporate these human-centric issues in design models into MDSE-based software code generators, enabling target software to dynamically adapt to differing user needs at run-time; and
5. use these human-centric requirements during software testing to support human-centric requirements-based testing of software systems, along with techniques to receive feedback from users on human factors-related defects in their software solutions.

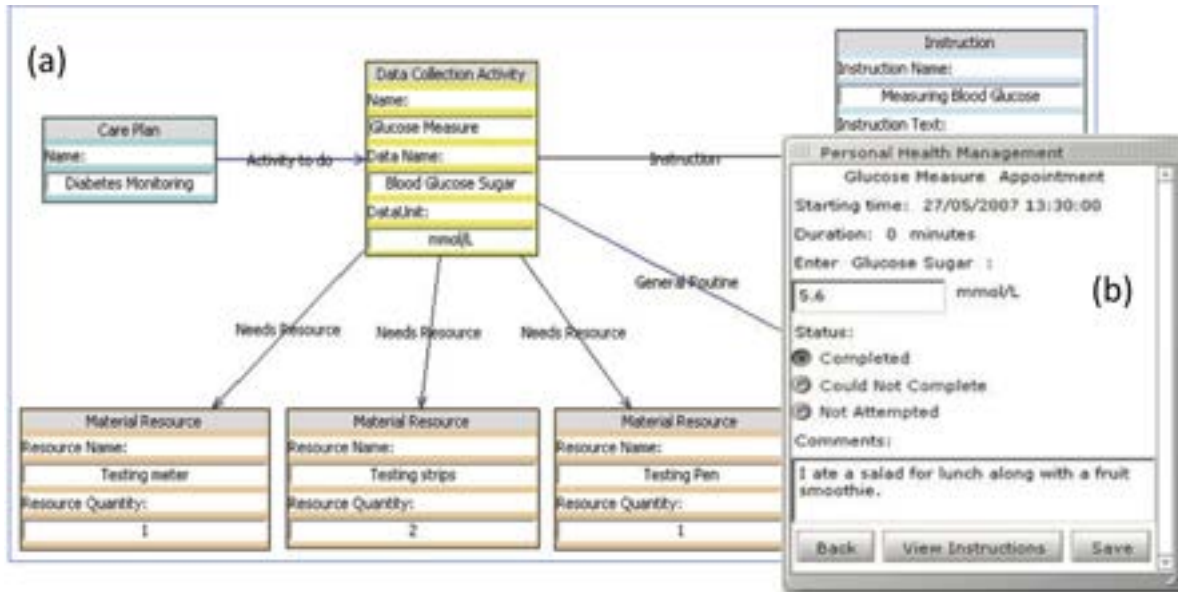


Figure 2: A clinician-oriented Domain-specific Visual Language for care plan modelling and using Model-driven Engineering to generate an eHealth app.

3 BACKGROUND

Model-Driven Software Engineering (MDSE) captures high-level models about software requirements i.e. what users need their software to do. MDSE then refines these models to detailed designs about how the software solution is organized, composed and its appearance. Model transformation then turns these models into software code. This is in contrast with most current software development methods which use informal and imprecise models, hand-translation into code via error-prone, time-consuming low-level hand-coding. Advantages of MDSE-based approaches include capture of formal models of a software system at high levels of abstraction, being able to formally reason about these high-level models and more quickly locate errors, and being able to generate lower-level software artefacts, such as code, without overheads and errors of traditional hand-translating informal models.

However, most MSDE approaches use generic requirements and design languages e.g. the Unified Modelling Language (UML) and extensions (Fontoura et al., 2000; Kent, 2002). These have the disadvantages of being overly complex and very difficult to use by non-software engineering domain experts (Hutchinson et al., 2011). Domain-specific Visual Languages (DSVLs) provide a more accessible approach to presenting complex models for domain experts (Sprinkle and Karsai, 2004). DSVLs use one or more visual metaphors, typically derived from the domain experts, to represent the model(s). They enable

domain experts to understand and even create and use the models directly, rather than rely on software engineers. These DSVLs are then used to generate software code and configuration artefacts to realise a software solution via MDSE approaches. This approach provides higher abstractions and productivity, improves target software quality, provides for repeatability, and supports systematic reuse of best practices (Sprinkle and Karsai, 2004; Hutchinson et al., 2011; Kent, 2002; Schmidt, 2006).

There are many DSVLs for MDSE tools (Sprinkle and Karsai, 2004; Li et al., 2014; Ali et al., 2013). A representative example is shown in Figure 2: (a) a custom DSVL designed for clinicians is being used to model a new patient care plan for diabetes and obesity management. Then (b) a model transformer takes the care plan and generates a mobile app to assist the patient to implement their plan (Khambati et al., 2008). This is a major improvement on developing software using conventional techniques. However, the approach fails to model or incorporate into the mobile app a range of critical human-centric issues, resulting in its failure in practice. Patient-specific, human-centric needs are not captured e.g. technology acceptance and emotional reactions e.g. some patients react negatively to the remote monitoring approach used. Some users are not English speakers. Colours and care plan model language used in the app are confusing for many older users. Users with eye-sight limitations find the app too hard to see and too fiddly to interact with. The app can't adapt to different contexts of use or preferences of the users e.g. it can't use their smart

home sensors or each patient's particular mobile app dialogue preferences. The app displays Euro-centric terminology about well-being, therefore, putting off some users from following their care plan. We need to incorporate these human-centric issues into MDE (Ameller et al., 2010).

There has been recent interest in the human-centric issues of complex software and how to better support these during software development. Agile methods, design thinking and living lab approaches all try and incorporate a human element both in eliciting software requirements and in involving end users of software in the development process (Dybå and Dingsøy, 2008; Hyysalo and Hakkarainen, 2014; Mummah et al., 2016). However, none capture human-centric issues in any systematic way and therefore the software fails to address several critical aspects of the human users. Some new approaches have tried to capture limited human-centric software issues. Emotional aspects of software usage include identifying the emotional reactions of users e.g. when engaging with health and fitness apps or for gaming. Work has been done modelling these Emotional Requirements and applying them to challenging eHealth domains (Miller et al., 2015; Curumsing et al., 2019).

Figure 3 shows a representative example using an emotion-oriented requirements DSVL to design better smart homes (Grundy et al., 2018). Here a conventional goal-based DSVL (1) has been augmented with a set of 'emotional goal' elements (2) specific to different users (3). Human characteristics like age, gender, culture and language can dramatically impact aspects of software, especially in the user interface presented by the software and the dialogue had with the user (Hartzel, 2003; Stock et al., 2008; Wirtz et al., 2009). Limited support for the capture of some of these has been developed. Another example is a multi-lingual requirements tool providing requirements modelling in English and Bahasa Malaysia, including supporting linguistic and some cultural differences between users (Kamalrudin et al., 2017). Usability and usability testing has long been studied in Human Computer Interaction (HCI) research and practice. However, usability defect reporting is very under-researched in the context of software engineering (Yusop et al., 2016). Similarly, a lot of work has been done on accessibility in HCI e.g. sight, hearing or cognitively impaired (Stock et al., 2008; Wirtz et al., 2009), and health IT e.g. mental health challenges when using mobile apps (Donker T, 2013). However, little has been done to evaluate the extent to which physical and mental challenges are properly addressed in engineering software development, and is also poorly supported in practice. Per-

sonality, team climate and organisational issues relating to people have been heavily researched in Management, Information Systems, and the personality of programmers and testers in software development (Pikkarainen et al., 2008; Soomro et al., 2016). However, little attention has been paid to how to go about supporting differing personality, team climate or organisational or user culture in software, nor to capture requirements relating to these human-centric issues. Traditional software requirements and design models have very limited (or no) ability to capture these sorts of human-centric software issues, and approaches are ad-hoc, inconsistent, and incomplete.

There are no modelling principles, DSVL-based model design principles, nor widely applicable, practical modelling tools to capture human-centric software issues at requirements or design levels. While a DSVL provides a more human-centric engineering approach, it fails to capture and support the key human-centric issues in the target software itself. Current MDSE tools, while providing significant software engineering benefits do not support modelling and using these critical human-centric issues. Current software engineering processes lack consistent, coherent ways to address these increasingly important human-centric software issues and thus they are often very incompletely supported or in fact are usually ignored.

4 APPROACH

We aim to employ several innovative approaches to (i) systematically capture and model a wide range of human-centric software requirements and develop a novel integrated taxonomy and formal model for these; (ii) promote a wide range of human-centric requirements for first-class consideration during software engineering by applying principles for modelling and reasoning about these human-centric requirements using DSVLs; (iii) support a wide range human-centric requirements in model-driven engineering during software generation and run-time reconfiguration via MDSE techniques; and (iv) systematically use human-centric requirements for requirements-based software testing and reporting human-centric software defects. This will improve model-driven software engineering by placing crucially important, but to date often forgotten, human-centric aspects of software as first-class considerations in model-driven software engineering. The critical importance of this is really only just becoming recognised, due to the increasing breadth of uses of IT in society and the increasing recognition that un-

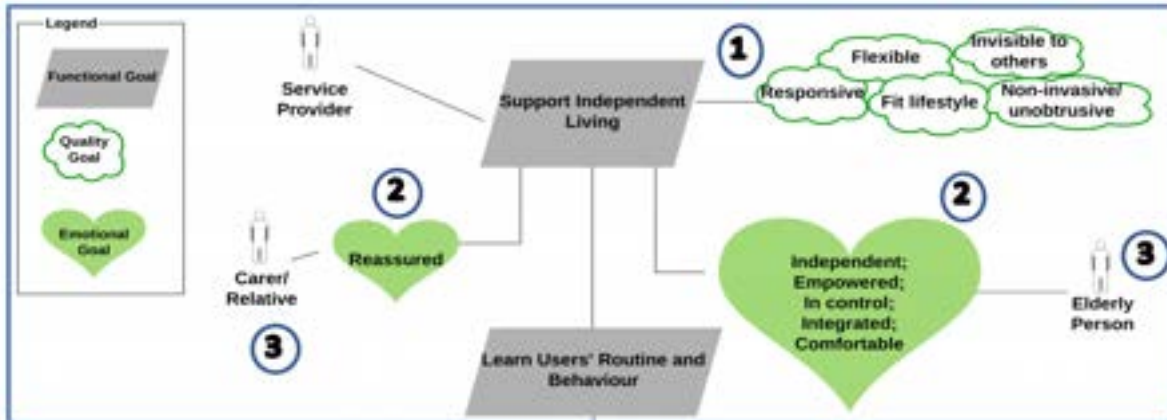


Figure 3: A Human-centric, Emotion-oriented Domain-specific Visual Language.

Understanding and incorporating the very diverse needs of our very diverse software end users is essential.

Key features of this approach include:

- Use of the Living Lab concept’s design thinking, agile, co-creation and continuous feedback mechanisms. These will be used to provide a MDSE approach in which human-centric requirements can be effectively and efficiently captured, treated as priorities by the software team, users can quickly report defective software violating these human-centric requirements, and the software team can work effectively with these end users to co-design changes.
- A set of principles for domain-specific visual modelling languages will be developed that enable software engineers to capture a wide range of human-centric aspects of software: including user’s age, gender, cultural preferences, language needs, emotional needs, personality and cognitive characteristics, and accessibility constraints, both physical and mental. These and other human end user characteristics are essential to prioritise during both software development and software deployment to ensure a useful and usable end product results for a broad range of end users.
- These principles will be used to design a range of novel DSVLs that fully support the capture of many of the important human-centric aspects and model them as the critical requirements issues that they are.
- Augmented models that ensure these modelled human-centric properties are preserved for use at design-time to ensure that MDSE-based solutions take them into account appropriately when generating software applications.
- We will develop a new framework for human-centric, requirements-based testing of software

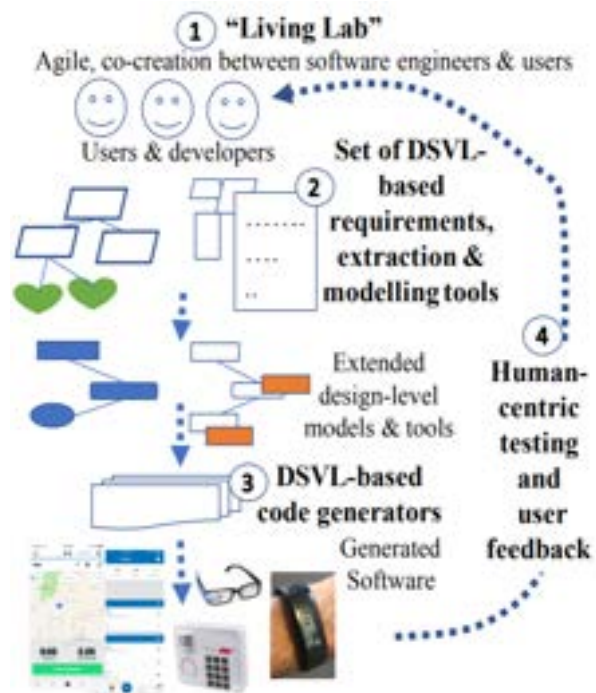


Figure 4: Overall approach

that can verify whether the constructed software systems meet these critical human-centric requirements.

5 RESEARCH ACTIVITIES

Figure 4 illustrates the new human-centric, model-driven software engineering approach we aim to produce. We have identified a set of key approaches that are needed to achieve this vision. An Agile Living Lab approach will be used to colocate the software team and target end users (Grundy et al., 2018). This will provide a co-creational environment to elicit

human-centric requirements, model and capture with human-centric DSVLs, and receive continuous feedback from users. A set of DSVL tools will be used to capture and model the human-centric requirements, validate them against design principles and best practice modelling patterns, and translate them to extended design-level models. A set of MDSE generators will generate software applications – code, configurations, etc. Unlike existing generators, these will take into account variations of end-users as specified in the human-centric requirements, producing either multiple versions of the target software applications and/or reconfigurable applications that adapt to each end user’s differing human-centric needs. A combination of human-centric requirements testing and continuous defect feedback will be fed to the development team. By leveraging the Living Lab concept, this will enable both faster feedback and defect correction, but also better evolution and modelling of the human-centric requirements over time. Lessons will be fed into the improvement of the DSVL tools, best practice patterns and MDSE generators. We have identified a set of necessary research activities to achieve this outcome, summarised below.

5.1 A Human-Centric Agile Living Lab

Human-centric requirements have to be elicited from target end users (or stakeholders), captured (or modelled) using our DSVL-based tools, used by extended MDSE solutions to generate software, and then the software tested and user feedback accepted and actioned to correct requirements and design model problems. A new approach is needed to effectively support the software team in achieving this, and propose investigating the Living Lab co-creation concept that has become popular in digital health software development (Hyysalo and Hakkarainen, 2014; Grundy et al., 2018). We are establishing this lab with a domain-specific focus with partner companies and target end users and the software team co-located as in Agile customer-in-team approaches (Dybå and Dingsøy, 2008; Pikkarainen et al., 2008). Target end users and developers closely collaborate to elicit, capture, test, use and refine the human-centric software requirements. The DSVL modelling tools, MDSE generators and testing tools all need to support collaborative capture, discussion and refinement of the human-centric requirements for this to be most effective. We plan to do this by extending our current work on developing digital health technologies (Grundy et al., 2018), human-centric software engineering processes in software teams, including person-ality and team climate (Salleh et al., 2018), and

collaborative DSVL-based modelling tools (Grundy et al., 2013).

To the best of our knowledge, no taxonomy of human-centric software requirements or even informal definition exists at this time. We are working on developing a new, rich taxonomy of human-centric requirements for software systems. The taxonomy will include different human-centric concepts relating to computer software, and will draw on other disciplines including HCI, usability, psychology, and others to build the conceptual model, and provide detailed relationships and trade-offs between different human-centric requirements. We are applying this to a number of representative requirements examples to test and refine it, and use the outcomes to inform the development of DSVLs, DSVL tools and MDSE solutions in other activities. This is critical research as it will provide software engineers a set of principles and conceptual model to model and reason about these kinds of requirements. We are conducting a detailed analysis of several representative real-world software applications from eHealth apps, smart homes, community service apps, educational apps, and other heavily human-centric requirements critical domains. From these we are developing a framework and model for prioritising human-centric software issues. This will characterise complex trade-offs and other relationships between different human-centric issues that make supporting one issue problematic for other issues, similar to the Cognitive Dimensions framework (Green and Petre, 1996). We will use focus groups with end users and developers to refine and validate our taxonomy. The taxonomy will be tested on real-world example requirements to gain feedback from both developers and end users to demonstrate its effectiveness. We are drawing on our extensive previous work developing taxonomies for design critics (Ali et al., 2013), emotion-oriented requirements (Curumsing et al., 2019), usability defects (Yusop et al., 2016), and team climate (Soomro et al., 2016).

5.2 Human-centric Requirements Engineering

While DSVLs have been an active research area for at least 20 years, remarkably few principles exist for design and evaluation of effective DSVLs (Moody, 2009). We are developing a set of new design principles and associated DSVL evaluation approaches to provide more rigorous principles and design steps for specifically human-centric DSVL development. This will require us to identify for the range of a human-centric software requirements and design is-

sues identified in the taxonomy built, how we can model these, use appropriate visual metaphors to represent the models, how we can support interaction with the visual models, and how we can reason about the suitability of these visual models in terms of usability and effectiveness. We are drawing upon the work on DSVL design tools to achieve this (Grundy et al., 2013; Li et al., 2014; Ali et al., 2013), as well as work on ‘Physics’ of Notations (Moody, 2009) and Cognitive Dimensions (Green and Petre, 1996) to develop these DSVL design principles for modelling human-centric software issues.

We are developing a range of new and augmented DSVLs to model a wide variety of human-centric issues at the requirements level for software systems. Some of these DSVLs extend existing requirements modelling languages – in successively more principled ways than currently – e.g. goal-directed requirements languages such as *i**, use cases and essential use cases, target user personas, user stories, etc. However, others may provide wholly novel requirements modelling techniques and diagrams that are then linked to other requirements models. We envisage novel requirements capture for things like identifying cultural, age, accessibility and personality aspects of target end users. Where multiple target end users for the same software application have differing human-centric requirements, multiple or composite models may be necessary. We are building on a wide range of DSVLs, including for design tools, requirements, reporting, business processes, surveys, performance testing, and many others (Ali et al., 2013; Grundy et al., 2013; Kamalrudin et al., 2017) as well as digital health software (Grundy et al., 2018) and work on modelling usability defects and emotional and multi-lingual requirements (Curumsing et al., 2019).

Software requirements need to be elicited from end users and these are typically held in a variety of documents and can be obtained in a variety of ways. We will work - in the context of the Living Lab - to develop new tools to extract human-centric requirements from diverse sources, including Powerpoint, Word, Excel, PDF, audio transcripts, images and video. A number of works have addressed different parts of this problem, including extracting requirements using light weight and heavy weight natural language processing (Ali et al., 2010; Lee and Xue, 1999). However, none have specifically addressed the extraction of a wide range of human-centric requirements. We will develop, trial and refine a set of extraction tools leveraging existing approaches but focused on human-centric requirements capture and representation using our DSVLs, within our living lab

approach, and leveraging our human-centric requirements taxonomy. These tools will also be refined as these other related activities are refined and extended, and applying these tools to representative real-world requirements artefacts will help us to test and extend the outputs of these other tasks. We will develop leading-edge tools for extracting requirements for goal-directed and multi-lingual models (Lee and Xue, 1999; Kamalrudin et al., 2017), and requirements checking and improvement (Ali et al., 2010).

5.3 Using Human-centric Issues in Model Driven Engineering

MDSE tools typically use Unified Modelling Language (UML) or similar design models. Even those using their own design models need to refine higher-level abstract requirements models into lower-level architectural, software design, interface, database and other models. We will explore different ways to effectively extend design-level models to capture necessary design-level human-centric properties, derived from higher level human-centric requirements-level properties (Ameller et al., 2010). For example, we want to capture design alternatives to achieve an application user interface for a target end user who has sight-impairment, prefers a gesture-based sensor interface to using a Smart phone, has limited mobility, and is quite “neurotic” about device feedback. We will evaluate the different modelling solutions via our living lab with both software engineers and end users, in terms of needed design information and preserving critical human-centred end user needs respectively. Even partial successful outcomes here will be immediately of interest and applicable for software teams. We will extend design models with aspects (Mouheeb et al., 2009), goal-use case model integration (Lee and Xue, 1999), and goal-models extended with emotions (Curumsing et al., 2019; Grundy et al., 2018). Just because we add human-centric issues to our requirements and design models does not mean they may be correct or even appropriate. We will develop a set of proactive tool support systems to advise software engineers of errors or potentially incorrect/unintended issues with their models (Ali et al., 2013; Robbins and Redmiles, 1998). This will enable the DSVL toolsets for human-centric requirements and design models to provide proactive feedback to modellers. To enable these design critics we will develop a range of “human-centric requirements and design patterns”. These will provide best-practice approaches to modelling complex requirements and design models mixed with human-centric issues. These features will be added to successive

iterations of our prototype tools from above. This work will build on our approaches to develop DSVL design critics (Ali et al., 2013) and DSVL-based requirements and design pattern modelling tools (Kamalrudin et al., 2011).

Once we have some quality design-level human-centric issues in models (incremental outcomes from the above activities), we can use these in model-driven engineering code and configuration generators. Results from this work will be fed back to extending the taxonomy and human-centric design principles. This will involve adding generators that consume design level models augmented with human-centric properties and synthesizing software applications that use these appropriately. For example, we might generate a gesture-based, passive-voice feedback solution for the target user from the example above. However, we might instead generate several interfaces for the same software feature, and at run-time configure the software either with pre-deployment knowledge, end user input, or even modify it while in use based on end user feedback. Thus for example a part of the software for our smart home example could adapt to different end users' current and changing needs (e.g. age, culture, emerging physical and mental challenges, personality etc). This work will be done incrementally, focusing on single issues first then looking at successively more complex combinations, adding support to the prototype tools and repeatedly trialling the tools. We will work on adding human-centric issues to MDSE code generators (Ameller et al., 2010), generating adaptive user interfaces (Lavie and Meyer, 2010), adaptive run-time software (Mohamed Almorisy, 2014), and DSVL-based MDSE solutions (Sprinkle and Karsai, 2004).

5.4 Using Human-centric Requirements in Testing, Defect Reporting and Continuous Feedback

Here we will address critically important issues of (i) testing whether the resultant software generated from our augmented MDSE actually meets the requirements specified; (ii) providing a feedback mechanism for end users to report defects in the software specifically relating to human-centric issues; and (iii) providing a feedback mechanism from software developers to users about changes made relating to their personal human-centric issues. We are developing a human-centric requirements-based testing framework, techniques and tools. These enable human-centric issues to be used in acceptance tests to improve validation of software against these require-

ments. We will also develop new human-centric defect reporting mechanisms and developer review and notification mechanisms. These will support continuous defect reporting, correction, and feedback via the living lab and remotely. Even partial outcomes would be of immediate benefit to the software engineering research and practice communities. This work is extending research on software tester practices and usability defect reporting (Yusop et al., 2016; Garousi and Zhi, 2013) and requirements-based testing (Kamalrudin et al., 2017).

5.5 Trials on Industry-scale Examples

We will trial our approach with real industry practitioners and organisations for whom human-centric issues are critical. Our approach is particularly suitable on applications with end users with challenging human-centric issues, such as physical or mental disability, English as second language, cognitive decline, very young or old, and needing software to adapt to their changing personal or contextual usage needs. Planned target application domains include digital health apps for community members, community educational apps, government service and transport apps and websites, and smart home and smart building management software.

6 CONCLUSION AND PERSPECTIVES

We have proposed a new approach to incorporating and supporting a wide range of human-centric issues into model-driven software development. We are setting up a co-creational, agile living lab in which to collaborative elicit user requirements, capture in augmented models, use to generate multiple solutions, and enable requirements-based testing and feedback. We are developing a new taxonomy of human-centric issues and then incorporating these into requirements and design modelling languages and tools. We are using these augmented models in model-driven engineering to generate multiple versions of software to support diverse end user needs, or support run-time adaptation of the generated code to these needs. We are supporting iterative user feedback to proactively incorporate human-centric issues into models and re-generating and re-deploying solutions.

ACKNOWLEDGEMENTS

Support from ARC Discovery Project DP170101932 and ARC Laureate Program FL190100035 is gratefully acknowledged

REFERENCES

- Ali, N. M., Hosking, J., and Grundy, J. (2013). A taxonomy and mapping of computer-based critiquing tools. *IEEE Transactions on Software Engineering*, 39(11):1494–1520.
- Ali, R., Dalpiaz, F., and Giorgini, P. (2010). A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458.
- Ameller, D., Franch, X., and Cabot, J. (2010). Dealing with non-functional requirements in model-driven development. In *2010 18th IEEE international requirements engineering conference*, pages 189–198. IEEE.
- Curumsing, M. K., Fernando, N., Abdelrazek, M., Vasa, R., Mouzakis, K., and Grundy, J. (2019). Emotion-oriented requirements engineering: A case study in developing a smart home system for the elderly. *Journal of Systems and Software*, 147:215 – 229.
- Donker T, Petrie K, P. J. C. J. B. M. C. H. (2013). Smartphones for smarter delivery of mental health programs: a systematic review. *J Med Internet Res*, 15(11).
- Dybå, T. and Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9):833 – 859.
- Fontoura, M., Pree, W., and Rumpe, B. (2000). *The Uml Profile for Framework Architectures*. Addison-Wesley Longman Publishing Co., Inc., USA.
- Garousi, V. and Zhi, J. (2013). A survey of software testing practices in canada. *Journal of Systems and Software*, 86(5):1354–1376.
- Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. *Journal of Visual Languages & Computing*, 7(2):131–174.
- Grundy, J., Mouzakis, K., Vasa, R., Cain, A., Curumsing, M., Abdelrazek, M., and Fernando, N. (2018). Supporting diverse challenges of ageing with digital enhanced living solutions. In *Global Telehealth Conference 2017*, pages 75–90. IOS Press.
- Grundy, J. C., Hosking, J., Li, K. N., Ali, N. M., Huh, J., and Li, R. L. (2013). Generating domain-specific visual language tools from abstract visual specifications. *IEEE Transactions on Software Engineering*, 39(4):487–515.
- Hartzel, K. (2003). How self-efficacy and gender issues affect software adoption and use. *Communications of the ACM*, 46(9):167–171.
- Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011). Empirical assessment of mde in industry. In *Proceedings of the 33rd international conference on software engineering*, pages 471–480.
- Hyysalo, S. and Hakkarainen, L. (2014). What difference does a living lab make? comparing two health technology innovation projects. *CoDesign*, 10(3-4):191–208.
- Kamalrudin, M., Hosking, J., and Grundy, J. (2011). Improving requirements quality using essential use case interaction patterns. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 531–540. IEEE.
- Kamalrudin, M., Hosking, J., and Grundy, J. (2017). Maramaic: tool support for consistency management and validation of requirements. *Automated software engineering*, 24(1):1–45.
- Kent, S. (2002). Model driven engineering. In *International Conference on Integrated Formal Methods*, pages 286–298. Springer.
- Khambati, A., Grundy, J., Warren, J., and Hosking, J. (2008). Model-driven development of mobile personal health care applications. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 467–470. IEEE.
- Lavie, T. and Meyer, J. (2010). Benefits and costs of adaptive user interfaces. *International Journal of Human-Computer Studies*, 68(8):508–524.
- Lee, J. and Xue, N.-L. (1999). Analyzing user requirements by use cases: A goal-driven approach. *IEEE software*, 16(4):92–101.
- Li, L., Grundy, J., and Hosking, J. (2014). A visual language and environment for enterprise system modelling and automation. *Journal of Visual Languages & Computing*, 25(4):253–277.
- Miller, T., Pedell, S., Lopez-Lorca, A. A., Mendoza, A., Sterling, L., and Keirman, A. (2015). Emotion-led modelling for people-oriented requirements engineering: the case study of emergency systems. *Journal of Systems and Software*, 105:54–71.
- Mohamed Almorsy, John Grundy, A. S. I. (2014). Adaptable, model-driven security engineering for saas cloud-based applications. *Automated software engineering*, 21(2):187–224.
- Moody, D. (2009). The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6):756–779.
- Mouheb, D., Talhi, C., Lima, V., Debbabi, M., Wang, L., and Pourzandi, M. (2009). Weaving security aspects into uml 2.0 design models. In *Proceedings of the 13th workshop on Aspect-oriented modeling*, pages 7–12.
- Mummah, S. A., Robinson, T. N., King, A. C., Gardner, C. D., and Sutton, S. (2016). Ideas (integrate, design, assess, and share): a framework and toolkit of strategies for the development of more effective digital interventions to change health behavior. *Journal of medical Internet research*, 18(12):e317.
- Pikkarainen, M., Haikara, J., Salo, O., Abrahamsson, P., and Still, J. (2008). The impact of agile practices on communication in software development. *Empirical Software Engineering*, 13(3):303–337.
- Prikladnicki, R., Dittrich, Y., Sharp, H., De Souza, C., Cataldo, M., and Hoda, R. (2013). Cooperative and

- human aspects of software engineering: Chase 2013. *SIGSOFT Softw. Eng. Notes*, 38(5):34–37.
- Robbins, J. E. and Redmiles, D. F. (1998). Software architecture critics in the argo design environment. *Knowledge-Based Systems*, 11(1):47–60.
- Salleh, N., Hoda, R., Su, M. T., Kanij, T., and Grundy, J. (2018). Recruitment, engagement and feedback in empirical software engineering studies in industrial contexts. *Information and software technology*, 98:161–172.
- Schmidt, D. C. (2006). Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25.
- Smith, J. (1998). *The Book*. The publishing company, London, 2nd edition.
- Soomro, A. B., Salleh, N., Mendes, E., Grundy, J., Burch, G., and Nordin, A. (2016). The effect of software engineers' personality traits on team climate and performance: A systematic literature review. *Information and Software Technology*, 73:52–65.
- Sprinkle, J. and Karsai, G. (2004). A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307.
- Stock, S. E., Davies, D. K., Wehmeyer, M. L., and Palmer, S. B. (2008). Evaluation of cognitively accessible software to increase independent access to cellphone technology for people with intellectual disability. *Journal of Intellectual Disability Research*, 52(12):1155–1164.
- Wirtz, S., Jakobs, E.-M., and Ziefle, M. (2009). Age-specific usability issues of software interfaces. In *Proceedings of the IEA*, volume 17.
- Yusop, N. S. M., Grundy, J., and Vasa, R. (2016). Reporting usability defects: A systematic literature review. *IEEE Transactions on Software Engineering*, 43(9):848–867.

THIS THESIS WAS TYPESET using \LaTeX , originally developed by Leslie Lamport and based on Donald Knuth's \TeX . The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.