KISS Workshop, Workshop at 2009 IEEE/ACM Automated Software Engineering Conference, Auckland, New Zealand, 17 Nov 2009.

A Generalised Event Handling Framework

Karen Li, John Hosking, John Grundy
Departments of Computer Science and Electrical and Computer Engineering,
University of Auckland,
Private Bag 92019, Auckland, New Zealand
{Karen, john, john-g}@cs.auckland.ac.nz

ABSTRACT

In earlier work we have developed three domain specific visual approaches for event-based system specification. The first, ViTABaL-WS, uses the Tool Abstraction (TA) metaphor to support specification of web services composition via higher level data and control flows and generation of BPEL4WS code. The second, Kaitiaki, uses an Event-Query-Filter-Action (EQFA) metaphor to allow visual primitives composition and java code generation for diagramming tool event handlers. The third, MaramaTatau, uses a spreadsheet-like metaphor to construct meta-model formulae visually to specify structural dependencies and constraints to be realised at runtime. We propose an integrated visual approach that is generalised from these three explored exemplar approaches to specify event handling behaviours. We derive a canonical event handling model which enables interoperability between these exemplar event models, with also the support for synthesised runtime visualisation. This paper discusses the requirements and design of the resulting general purpose event handling framework, its evaluation and some key future directions.

1. INTRODUCTION

The event-driven paradigm is widely used in a range of application domains due to its flexibility for constructing dynamic system interactions. Visual approaches, compared to custom code writing, have shown their advantages in minimising design and implementation effort and improving understandability of programs [1, 2, 4, 6, 10]. This suggests that a visual language that supports event integration specification is likely to be a positive approach for the design and construction of a complex event-based system. Visualisation support (tool support) for the event propagations in a running system is also necessary in order to allow users to track and control the system execution behaviour [5, 9]. Using different high-level visual metaphors for event-handling support and providing backend processing tool support for event integration specification were our main objectives in the work presented here. Based on in-depth research on current event handling techniques including custom scripting, constraint-based programming models and meta-model tool event handling metaphors, our initial goal was to develop three exemplar domain-specific visual languages to examine event handler specification issues in different domains at different level of abstractions. We have previously described each of these exemplars [13-15]. ViTABaL-WS was developed for the event-based web services composition domain to provide a high-level visual language for the design and construction of Tool Abstraction action-event-based architecture. Kaitiaki was developed for diagramming-based design tools event handling domain to provide an intermediate level extensible Event-Query-Filter-Action language for responding to propagated events. MaramaTatau was developed to look at visual declarative constraint specifications that map to meta-model elements using OCL with a simple spreadsheet-like interface, and it has an implementation level abstraction. The subsequent goal was to generalise from them to a visual metaphor and an environment for specifying general event handling integration. In achieving this our aim was that the general visual metaphor should be able to adapt the event-based communication model to a wide range of application domains, and also support complex and interactive system design and implementation.

Roberts and Johnson proposed a set of patterns that are used together as a pattern language for developing and evolving object-oriented frameworks [18]. The leading pattern is *Three Examples*, which denotes that abstractions can be well developed by generalising from concrete examples. The pattern suggests that three examples should be initially used, in either succession or parallel, to establish a framework by identifying common, reusable abstractions, and more examples are to be explored to make the framework more general. This is used as our basis for generalising our event handling frameworks.

The EASY (Event Abstraction SYstem) framework [8] was a prior attempt to generalise a unified event-based software architecture from the synthesis of a set of event handling elements defined in Change Propagation and Response Graphs (CPRGs) [7], ViTABaL [5] and Serendipity [6]. CPRGs can effectively describe state-change events and the structural aspects of event-based software architectures. ViTABaL supports visual representation of propagations of action events between software components. Serendipity allows event filtering and response mechanisms to be specified in a graphical way. EASY unifies the handling of CPRGs' state-change events and ViTABaL's action events by incorporating Serendipity's event response abilities. The advantages of the three visual languages, including their visual description capabilities for both structural aspects and dynamic behaviours of event-based architectures, are combined to provide a more general architecture description language that supports wider-ranging event-based architecture design and implementation. Figure 1 shows an EASY example, which has CPRGs' components and relationships as the backbone, ViTABaL's specification of data and toolie interconnectivity, and Serendipity's specification of event handling using filters and actions.

We aimed to generalise our event integration framework using the Evolving Framework Pattern Language approach [18] and following the EASY [8] framework as an example. In this paper, we begin by revisiting the three exemplars to introduce some context for the following discussion. We then describe the requirements and our high level approach before examining the details of generalisation. We then describe our evaluation techniques and conclude with a reiteration of current research directions.

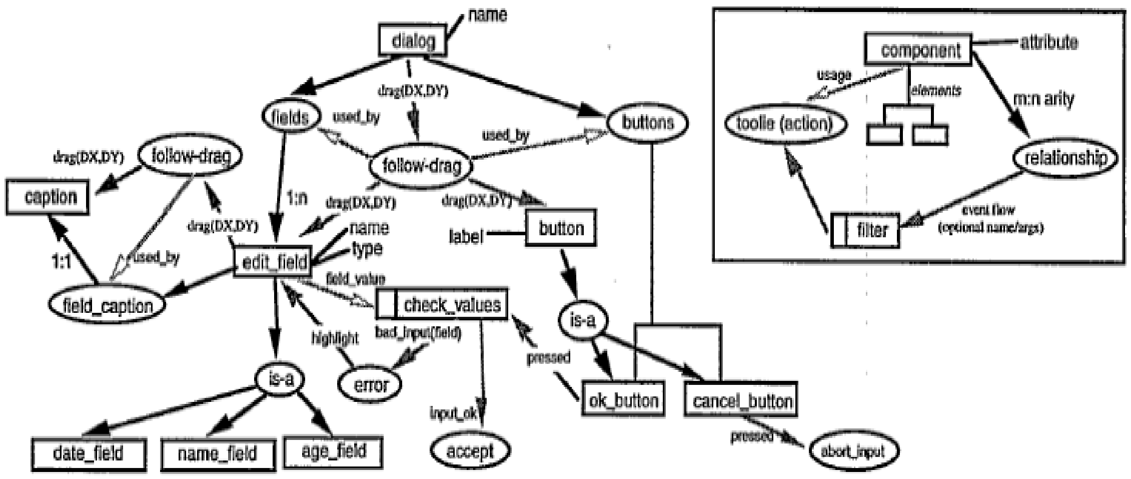


Figure 1. Merging CPRGs organization, ViTABaL event propagation and Serendipity event filtering/action. [8]

2. THE THREE EXAMPLES

Our three examples looked at visual metaphoric specifications of event handling behaviours in the areas of web services and business process composition, GUI event handling, and constraint-based meta-modelling. We briefly review the event propagation model features and the building blocks defined for each of ViTABaL-WS, Kaitiaki and MaramaTatau, to introduce some context before we illustrate our design of generalising them to a common set of primitives and abstractions.

ViTABaL-WS uses a Tool Abstraction [5] metaphor for describing relationships between service definitions in multiple views. It supports modelling of complex interactions between web service components, plus code generation and visualisation of running systems.

ViTABaL-WS model views describe the interconnections between toolies and abstract data structures (ADSs). These interconnections can be annotated with different type of data flow, control flow, and event flow connections. Different kinds of subscribe-notify event propagations including one way broadcast, request-response, listen-before and listen-after can be used between the connected toolies and ADSs. Toolies encapsulate behaviour in that they respond to events to carry out some system function. ADSs encapsulate data and respond to events to store, retrieve or modify data [5, 13].

Modified toolies or ADSs broadcast to all their inter-connected components about the change. Receiving components interpret the change descriptions and modify their state or execute actions accordingly with possible further change descriptions to be generated [5]. ViTABaL provides an architecture description language for the event-based Tool Abstraction paradigm. ViTABaL-WS includes a few more building blocks to control event-based behaviour by specifying roles, sequences, decisions, type transformations, iterations, and transactions. Figure 2 shows an exemplar ViTABaL-WS event propagation view (generated in Marama [10]) that specifies a set of subscribe-notify event propagations between toolies (Marama library functions) and ADSs (Marama shared data structures).

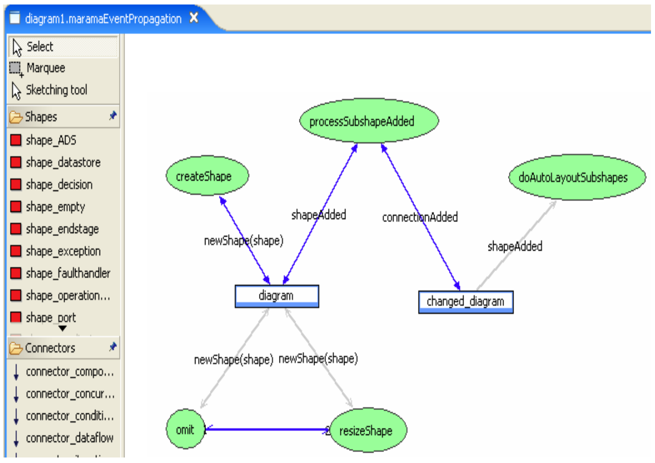


Figure 2. ViTABaL-WS event propagation specification.

Kaitiaki provides end users ways to express event handling mechanisms via visual specifications. It uses an "Event-Query-Filter-Action" metaphor for describing behaviours for diagramming-based design tools and multiple-views of data flow in a modelled process. Kaitiaki supports building up complex event handlers in parts, providing the representation of:

- key "building blocks" of state query, data filtering and state modification,
- event objects and their attributes,
- data propagation between event, query, filter and action representations, and
- iteration and conditional data flow

While ViTABaL-WS visually describes only the event-based inter-connections between abstract components with the lack of event responses, Kaitiaki's events, filters, queries and actions provide a visual design level notation for specifying event handling mechanisms.

Kaitiaki provides graphical views for specifying handling of both built-in and customised state-change and action events via queries, filters and actions. Queries select data from a common model repository. Filters apply pattern-matching to incoming data, passing matching data to other queries/filters/actions. Actions execute operations which may modify incoming data, display information, or generate new events. Concrete end user domain icons can be added to mitigate the abstraction and make the specification more readable. Figure 3 shows an exemplar Kaitiaki event handler specification (generated in Marama) for aligning diagram shapes (a) and its runtime execution effect (b). The handler responds to a "shapeAdded" event, filters out the "TableShape", and then aligns the newly added "TableShape" with the existing ones that are queried from the diagram.

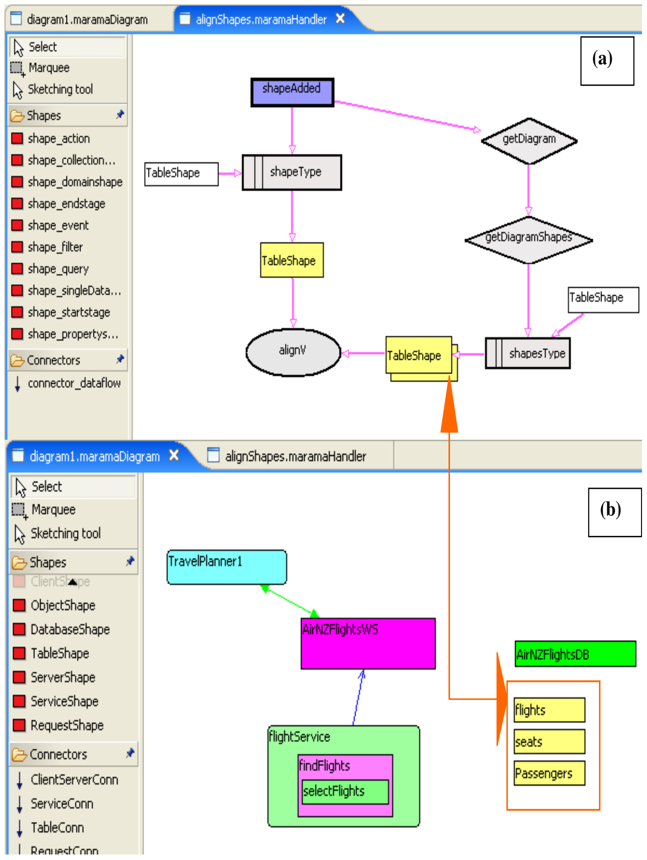
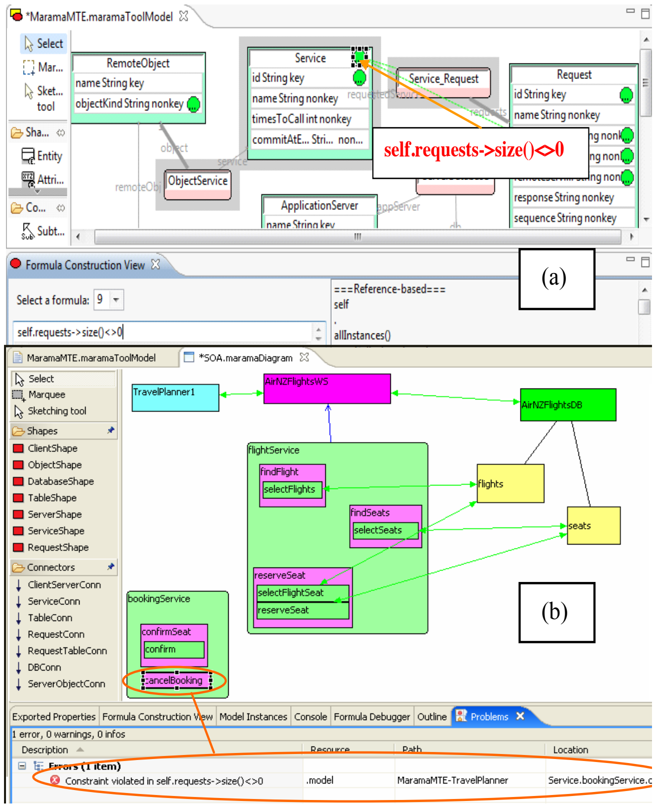


Figure 3. Kaitiaki event handler specification (a) and its runtime execution effect (b).

The Marama framework [10] provides Eclipse-based editors for Pounamu [19] generated domain-specific modelling environments. It contains a set of classes, attributes, methods, relationships and events. The framework is packaged into three major parts: model, editor and reusable handlers, in the style of a model-view-controller based separation of implementation concerns. These reusable framework elements support easy creation, modification and extension for building domain-specific modelling environments.

Marama provides a rich structural notation for specifying tool architecture/meta-model via an entity-relationship mechanism. MaramaTatau is used to specify value dependencies and modelling constraints upon these Marama structural specifications. Though was initially designed for constraining entity-relationship based meta-models, MaramaTatau has evolved to be usable with any Marama view type specifications.

MaramaTatau uses a declarative spreadsheet-like approach to construct meta-model formulae to extend behaviour specification of visual design tools including the specification of property-change event handling and executable query/action constraints. Value dependencies and modelling constraints are state-change events to be handled in MaramaTatau via a uni-directional change-propagation with side-effect extensions to dependent components. A formula is constructed visually by clicking on entity-relationship meta-model elements (i.e. entity type, association type, and attribute) and a list of library provided OCL-based functions. Dependency links are added to annotate explicitly the relationships of inter-dependent elements. Formula construction is similar to a spreadsheet but expressed at a type rather than an instance level. The visually specified values at meta-model level are propagated from sources to dependent targets and interpreted at runtime in selected model views. Figure 4 demonstrates its use in specifying an entity invariant constraint.



1 error, 0 warnings, 0 infos

Description	Resource	Path	Location
Errors (1 item)			
Constraint violated in self.requests->size()<>0	.model	MaramaMTE-TravelPlanner	Service.bookingService.c

Figure 4. MaramaTatau constrain specification (a) and its runtime execution effect (b).

In Figure 4 (a) we have extended a service-oriented architecture model with a constraint specifying that every service instance must serve at least one service request. This is expressed as a constraint on the Service entity, with OCL expression "`self.requests->size()<>0`", shown in the overlay. When this formula evaluates false for a service in a model instance, e.g. the "cancelBooking" service of the "bookingService" remote object in Figure 4 (b), a constraint violation error is generated, with a problem marker representation appeared in the Eclipse Problems view (shown below) to provide the user details of the violated constraint. In this case, to solve the identified error, the user needs to add a Request entity for the identified service. When this is done, the constraint evaluates to true and the constraint error is removed from the Problems view.

3. REQUIREMENTS AND APPROACH

The generalisation of the integrated event handling framework requires a variety of specialised modules to contribute to the framework capability and complexity. We have identified similarities in our three examples, and these include a set of event handling modules and the representations of data flows and event dependencies among their visual building blocks. The similarities can be generalised to a common model representation to allow better reuse and easier extension. Some event handlers can be specified in multiple ways using ViTABaL-WS, Kaitiaki or MaramaTatau. Users may choose to use their favoured metaphor and may also combine their specifications in multiple ways. Multiple tools are useful for developers in that they provide abstractions for separately and progressively modelling a software system using different views and representations. The developers can specify the structure and behaviour of a model in parts then integrate them to generate dynamic environments with various constraints enforced.

A general purpose event handling framework should provide reusable design and implementation for a wide-range of event-based applications. From our previous work, we have identified common issues in our current event handling specification and visualisation techniques. From this, we elaborate a set of requirements for the generalisation of our event handling integration framework:

- The generalised framework should incorporate compositional primitives as building blocks and different communication relationships between them. It also should contain mapping/integration schemes as a crossover between ViTABaL-WS, Kaitiaki and MaramaTatau, and possibly other limited-domain event handling models in the future.
- The common model representation needs to be identified from the specialised modules from ViTABaL-WS, Kaitiaki and MaramaTatau. The relationships among the modules need to be established so that the modules can collaborate with one another. Duplications need to be removed so that the common model is redundancy free.
- The generalised framework needs to offer graphical notations in the style of the three metaphoric exemplars, together with additional textual notations to allow users to escape to code when specifying complex custom behaviours such as code generation.
- The generalised integrated framework must contain reusable designs to allow users to initialise their system and should allow users to specify customised event types, event generators, event receivers and event handling building blocks to enhance the extensibility and flexibility of the framework.
- Multiple views of data, event and behaviour representations must be kept consistent at both the model and user interface levels to ensure the correctness of generated environments.
- The generalised framework should support further tool integration via a canonical data/event model extension and consistent user interfaces.
- The generalised framework should provide mechanisms to allow easy navigation from one view of the specification to another.
- Though visual languages are more self descriptive than textual languages, the framework should still provide support for detailed documentation of modelling elements.
- The generalised framework should allow event propagations to be traced and event handling results to be visualised in running systems based on a user interactive visual debugging model.

Our generalisation approach employs the Evolving Frameworks Pattern Language [18]. By abstracting from the three earlier, limited-domain exemplars, a general meta-model representation that combines atomic primitives (either shared or non-shared) extended by the three visual languages is defined. This common model supports multiple metaphoric views in the style of the three exemplars and will support generation to a range of underlying implementation technologies for execution or interpretation.

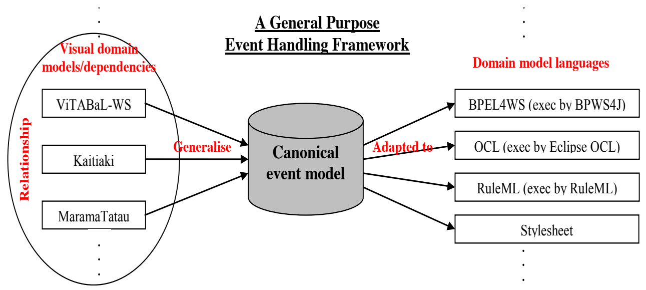


Figure 5. A general purpose event handling framework.

The generalisation method that we exploit can be illustrated in Figure 5. The three visual domain models ViTABaL-WS, Kaitiaki and MaramaTatau on the left of Figure 5 can be integrated based on their metaphorical supplement and their common abstraction and dependency relationships. ViTABaL-WS's Tool Abstraction metaphor is used to define high-level abstract data and functions and their coordination, where abstract data is further constrained using MaramaTatau's spreadsheet metaphor, and abstract functions are further refined via Kaitiaki's Event-Query-Filter-Action metaphor. The integration of the three metaphors is analogous to the desktop Windowing metaphor that is associated to the both Folder and Tree metaphors. Similar data could be represented in different ways (e.g. Folder and Tree), but maintained in the same highly abstract umbrella representation (e.g. Windows). As a consequence, they can generalise to a common event model representation (as seen in the middle of Figure 5). The canonical event model can then be mapped or adapted to a range of domain model implementation languages to be executed (e.g. IBM's BPEL4WS, OMG's OCL, and RuleML [16] and stylesheet as seen on the right of Figure 5) using appropriate domain engines. The object-oriented framework is readily extensible so more event-based domain models and their dependencies can be added in the future. Our immediate next example being planned is the OMG's Business Process Modelling Notation (BPMN) in the enterprise modelling domain. The new models to integrate can reuse the canonical model's components through inheritance or composition, and can add more features and support to evolve the framework.

In order to derive a suitable common model we need to be able to represent all of the concepts from the three examples. We also need a way to map between related concepts in each metaphor. This common model supports multiple metaphoric views in the style of the three exemplars and thus is in multiple paradigms.

4. GENERALISATION

In this section, we discuss the design of the resulting general purpose event handling framework obtained by generalising from the three exemplar approaches described in earlier work [13-15]. Synthesised program visualisation and framework evolution are also described here to show the integrated runtime features and framework extendibility.

4.1 The Generalised Event Handling Framework

To generalise our work on ViTABaL-WS, Kaitiaki, and MaramaTatau, we have designed a set of Marama meta-tools to provide a better platform and a vehicle for allowing us to explore event-handling integration. Figure 6 illustrates the Marama meta-tools approach, which is an add-on to the Marama framework that includes five sub-tools: Meta-model Definer, Shape Designer, View Type Definer, Event Propagation Definer and Visual Event Handler Definer. The incorporative use of these sub-tools facilitates easy event-based behavioural modelling and integration that is unified with system structural modelling.

Entities, relationships, shapes, connectors and mappings form the structural backbone of the meta-modelling framework in Marama meta-tools. A sub-model for the event handling building blocks is added on top of the Marama EMF model to support specifying event-based manipulations of Marama structural model elements. The behavioural sub-model contains the definition of all the generalised canonical event model elements and provides a structured way to query and update these element instances. The behavioural sub-model is represented in different metaphoric views in the style of ViTABaL-WS, Kaitiaki and MaramaTatau. MaramaTatau specifies inter-dependency of Marama static modelling components by adding formulae over both model and view data structures. A one-way constraint system is exploited to compute dependent values at runtime during tool usage. ViTABaL-WS specifies event propagations and other inter-connectivity of toolies (user or library functions) and their shared ADS pool (Marama structural components). The event representations are propagated to the listening toolies which match them to the event patterns they respond to, and the response is invoked [5]. Kaitiaki specifies detailed toolie responses via event propagations through a set of library-defined pattern matching queries, filters and actions.

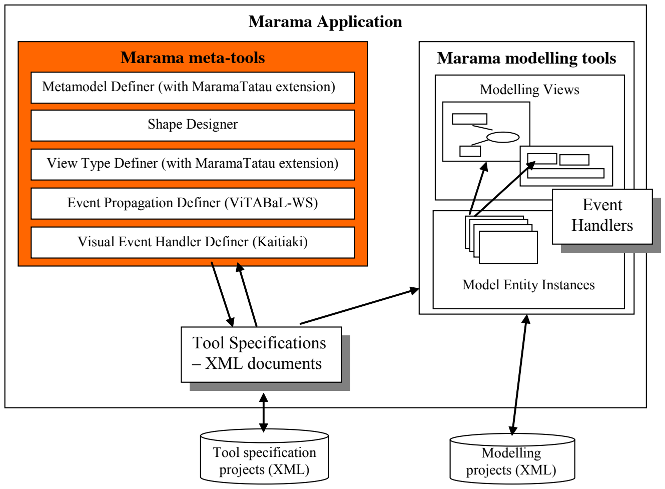

Marama Application
Marama meta-tools
Metamodel Definer (with MaramaTatau extension)
Shape Designer
View Type Definer (with MaramaTatau extension)
Event Propagation Definer (ViTABaL-WS)
Visual Event Handler Definer (Kaitiaki)
Marama modelling tools
Modelling Views
Event Handlers
Model Entity Instances
Tool Specifications – XML documents
Tool specification projects (XML)
Modelling projects (XML)

Figure 6. The Marama Meta-tools approach.

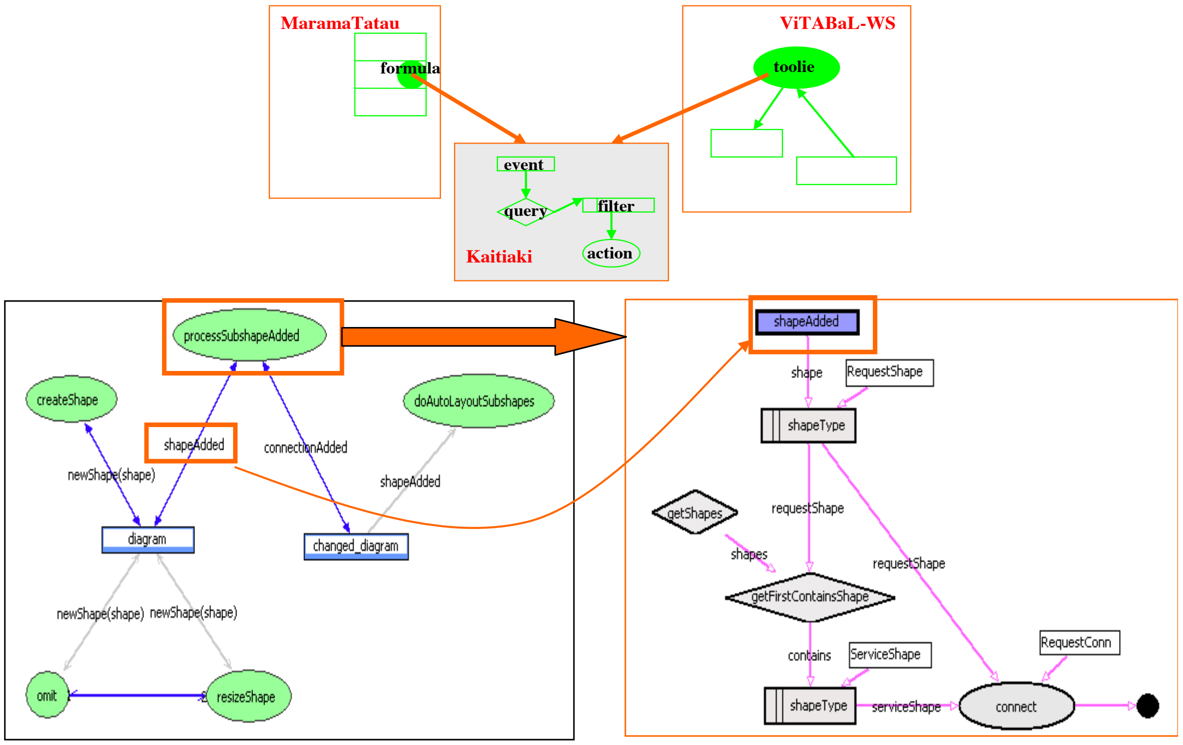


Figure 7. Unified event handling in MaramaTatau and ViTABaL-WS using Kaitiaki.

As in the EASY framework [8], Marama meta-tools also permit MaramaTatau state-change events and ViTABaL-WS action events to be handled in a unified manner, via event response modelling capabilities of Kaitiaki as illustrated on the top of Figure 7. A detailed example is also provided in Figure 7, where in the ViTABaL-WS diagram on the lower left a "shapeAdded" event propagates from the data structure "diagram" to the toolie "processSubshapeAdded" which is an event handler further defined in the Kaitiaki view on the lower right. MaramaTatau state-change events can also be handled in this extensible manner, via Kaitiaki specifications. This aims to maintain the advantage of MaramaTatau of effective structural dependency and constraints specification, and that of ViTABaL-WS and Kaitiaki of visual representation of event propagation and response mechanisms, while also providing user-defined behaviour extension of MaramaTatau and integrating the three languages to provide unified specifications.

Each of ViTABaL-WS, Kaitiaki and MaramaTatau has their own strengths in handling events. They are mainly complementary to one another instead of overlapping. However, there exists the possibility to specify an event handler in multiple ways using ViTABaL-WS, Kaitiaki or MaramaTatau, though one specification may not be as efficient as the other, the required event handling effect can still be achieved. The connectivity and inter-changeability of the different metaphoric specifications can facilitate mapping concepts in each metaphor and thus provide effective demonstration and model checking. To allow one specification to generate others with corresponding implementation classes, a set of mapping schemas can be defined in MaramaTorua [12] to provide interchanging mechanisms between ViTABaL-WS, Kaitiaki and MaramaTatau specifications. We are currently exploring the mapping specifications to be used for such integration.

The canonical event handling model (behaviour sub-model of Marama meta-tools) enables development of general purpose event-based system specifications. The meta-model elements from the three visual languages have been combined with redundancies removed and some bridging elements added. An excerpt of the component library of the event handling abstractions framework is illustrated in Figure 8, where mappings of the model elements to those used in the three visual languages are also indicated via coloured and patterned boxes. The component library mainly includes the relationships between event, event generator, event service, event listener and event handler elements. The event handler is further sub-typed including publish, subscribe-notify, invoke activity, generate event, capture event and custom handler. The connectivity types supported in the framework include structural generalisation, association and composition, and behavioural control, data and event flows. The CompoundActivity interface may take multiple possible roles as event generator, event listener or event handler, may contain the ViTABaL-WS, Kaitiaki and MaramaTatau building blocks and may be involved in a variety of data manipulation and dynamic connectivity operations.

Almost all elements in this common model are defined as extensible. Particular hot spots, or places in the architecture where adaptations for specific functionality should be made [18], include:

- Event - The framework supports a set of system events together with user-defined custom events to be added by either specifying new event details or by sub-typing/composing existing event types.
- Event handler - The framework supports a set of system event handler building blocks together with user-defined custom event handlers to be added by either specifying new event handler details or by sub-typing/composing existing event handler building blocks. For examples, the event handler types of a GUI system can include additional "UpdateUI" handler, "AutoLayout" handler, "PromptMessage" handler etc.
- Control flows - Control flows can be stereotypes to specify the transition time requirement (`<<synch>>` or `<<asynch>>`), the transition sequence (`<<1.1.2>>`, `<<StartWith>>`, `<<EndWith>>`), etc. Concurrent transitions do not need to be explicitly modelled. When the condition of a transition is met, the transition is invoked immediately. So when an element is associated with multiple transitions, the transitions are concurrent when their conditions are satisfied at the same point of time.

Some bridging elements are introduced to the common event handling model to facilitate abstraction inheritance and crossover among the metaphoric building blocks. These typically include Event Service, Event Listener, Event Handler, Connectivity, Event, Object and Compound Activity (see Figure 8). The event service, acting as a registrant, receives all notifications (e.g. entity changed) and forwards them to any associated event behavioural views – ViTABaL-WS, Kaitiaki or MaramaTatau. Inter-communications of the three behavioural views are monitored by the event service and automatically delegated to Marama processing components.

This canonical model representation is used to instantiate behaviour specifications in ViTABaL-WS, Kaitiaki and MaramaTatau views such as those shown in Figure 2, Figure 3 and Figure 4, and interoperability is achieved via reference relationship configurations (using the property sheet to select an element to refer to) and navigations (double-clicking a referring element in one view to navigate to the referred element in another view) among behavioural elements and views. For instance, a toolie in a ViTABaL-WS view can refer to a complete Kaitiaki event handler specification or an individual query/filter/action in the view; a Kaitiaki query/filter/action can refer to a MaramaTatau formula and vice versa. The behaviour model instances are analysed at specification time and are used to generate event handler code to be executed at runtime. Such a canonical representation of event handling specifications enables further behavioural paradigms to be easily integrated into the framework.

4.2 Program Visualisation

The model-view-controller pattern is used in Marama meta-tools to synthesise event-based behaviour from multiple views. The three distinctive yet collaborative metaphoric views generalise to a common model implemented in the event handling abstraction framework which in turn accesses class libraries and then interprets them to query and update Marama EMF model and view representations. Visualisation of dynamic event handling behaviour is achieved using a similar model-view-controller approach, where runtime behaviour model states are used to animate the associated diagram elements. The user has full control of running the animation, stepping into the next invocation of a building block and viewing query results or state changes.

Marama meta-tools allow users to visualise tool specifications and their executions reusing their metaphoric modelling views, to provide system information at the right abstraction level.

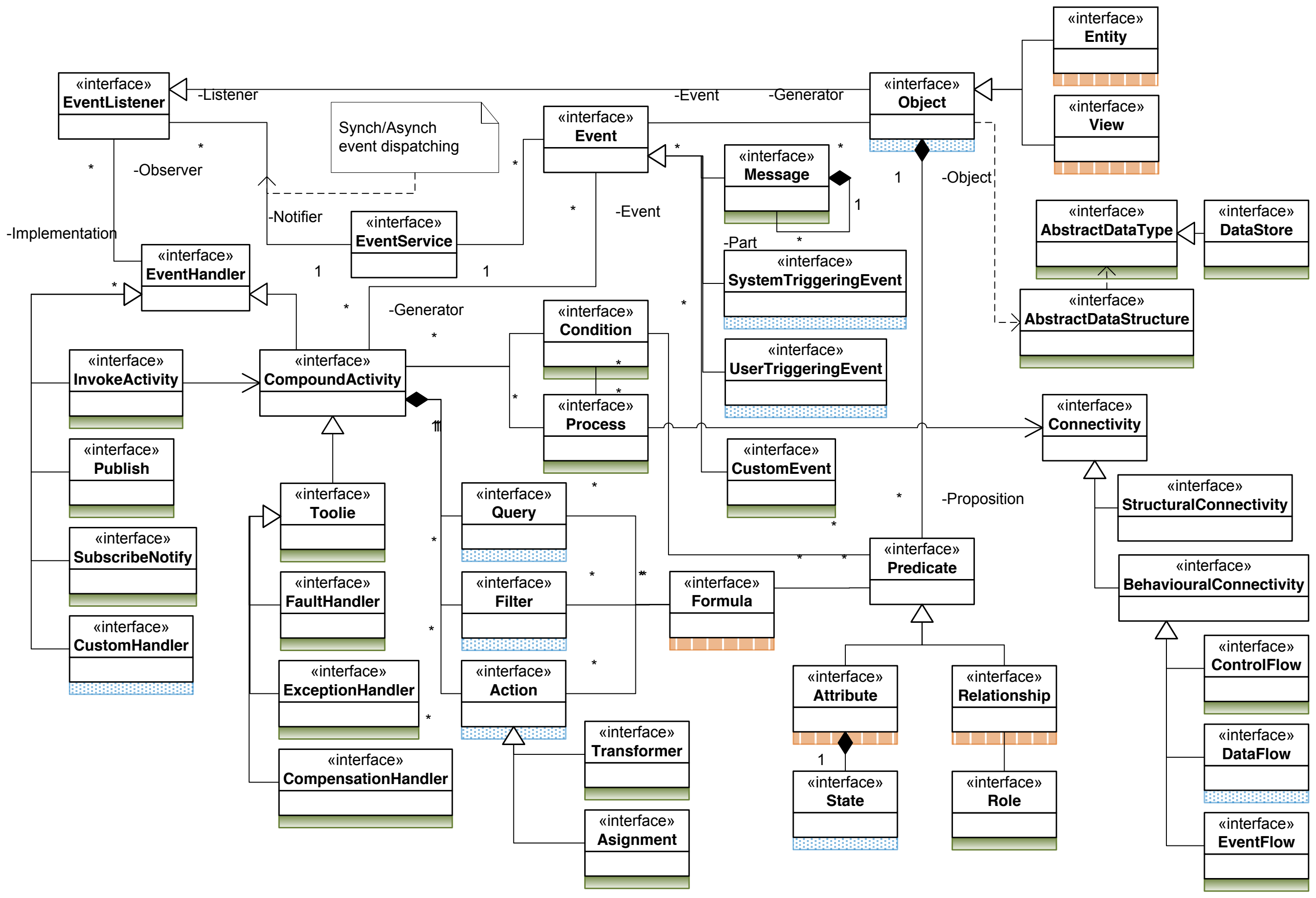


Figure 8. Common event handling model (🟩 for ViTABaL-WS Components; ▨ for Kaitiaki components; 🟧 for MaramaTatau components).

We exploit the debugging service instrumentation mechanism [13] initially exploited in ViTABaL-WS to generate low-level tracing events on modelling elements. The Marama meta-tools framework 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, and supplies dynamic state information to the Marama Visual Debugger. A specialised debugging and inspection tool is used to allow execution state of event-based systems to be queried, visualised and dynamically modified. The debugger tool 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.

The individual "debug and step into" visualisation of ViTABaL-WS, Kaitiaki and MaramaTatau are combined to allow cooperative invocation and step-by-step visualisation of execution results at the point of execution of each building block in a particular view. Figure 9 illustrates the visualisation of runtime interpreted formulae (a) followed by an event handler (b) on a Marama model. The Meta-model Definer view and the Visual Event Handler Definer view with the respective formula and event handler specifications are juxtaposed with the runtime Marama model view. From the Visual Debugger, user has the control over the execution of a behaviour building block interpretation. Once the behaviour is interpreted, the affected runtime model element is annotated (with the 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.

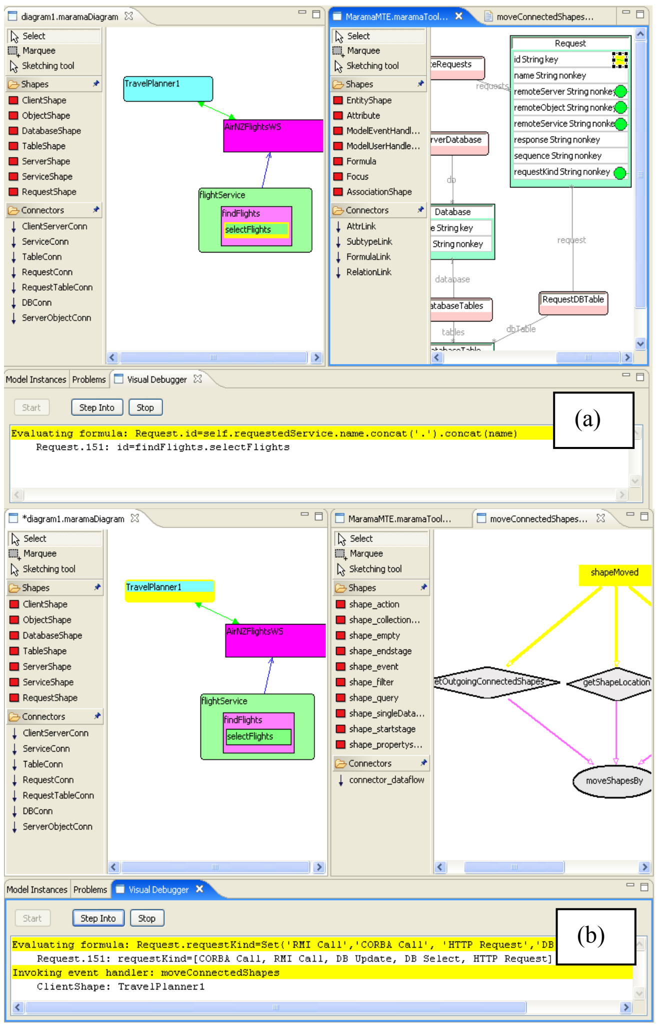
Run-time monitoring of the Marama meta-tools for performance analysis could be supported via the visual debugging sub-system. The visual debugger could be further enhanced with "watch" controls so that the user can choose to trace a certain event and its response instead of debugging the entire behavioural specification.

4.3 Framework Evolution

The event handling abstractions framework in Marama meta-tools is both black-box and white-box. It provides reuse by both inheritance and composition. Based on the evolving frameworks pattern language [18], our framework will be evolved by abstracting from additional examples to make it more general in the future.

Subsequent exemplars are to be developed based on the white-box framework. In generalising more of the event handling abstraction framework by integrating a further exemplar, we will first examine what abstractions from the canonical model's component library (as shown in Figure 8) can be reused (through either inheritance or composition) by the new domain model, and then examine what new features and support can be added to evolve the framework.

The integration of MaramaTorua [12] can also provide another view type, with event-driven mechanisms to allow translation of one event handler view to another (e.g. generating the event handlers/formulae to keep view/model consistent from MaramaTorua specifications) to be specified internally and automatically without the need to invoke it as an independent non-event-driven third-party tool. Our generalised model should support new metaphors/models to be further sub-typed or composed.



```
Evaluating formula: Request.id=self.requestedService.name.concat('.').concat(name)
    Request.151: id=findFlights.selectFlights
```

```
Evaluating formula: Request.requestKind=Set{'RMI Call','CORBA Call', 'HTTP Request','DB
    Request.151: requestKind=[CORBA Call, RMI Call, DB Update, DB Select, HTTP Request]
Invoking event handler: moveConnectedShapes
    ClientShape: TravelPlanner1
```

Figure 9. Visual debugging MaramaTatau formulae (a) followed by a Kaitiaki event handler (b).

5. EVALUATION

Following our initial prototype development, we have conducted both developer-based and end user-based evaluations of the Marama meta-tools in general to test their usability and effectiveness for specifying event-based system integration with the aim of identifying potential problems. The evaluation results have been sufficiently positive for us to release the Marama meta-tools as a publicly accessible toolset following a number of enhancements to address tool stability.

It is not a straightforward task to evaluate a substantial environment/toolset such as the Marama meta-tools, as it involves multiple points of views of tool developer, end users of developed tool, usability, utility, etc. [19]). Most formal usability evaluation approaches are limited to understanding the effect of one or two variables [3, 11]. Controlling for variability is an almost impossible undertaking when assessing the usability of a large environment. Formal evaluation for this type of system is hard. This means we have had to adopt a variety of less formal, but overlapping approaches to obtain usability and efficacy data.

We have evaluated Marama meta-tools at several levels and through a variety of mechanisms in a similar way that evaluations of our Pounamu metatool were conducted [19]. These include:

- We, the designers, conducted a cognitive dimensions [4] evaluation focusing on the event handling specifications. Cognitive dimensions allow us to understand usability tradeoffs and hence where mitigations need to be placed. Each of the three individual languages and environments feature easy and effective specifications with some dimensional tradeoffs where we have placed effort to provide mitigations (e.g. to minimise hidden dependency issues in MaramaTatau). The use of three distinct metaphors together in the system has increased the abstraction gradient and the initial learning curve of the Marama meta-tools, but provided effective and consistent event handler specifications by addressing identified concerns and allowing tool designers to escape from writing conventional code.
- We, the designers, conducted an evaluation of the Marama meta-tools along with the event handling abstraction framework against the requirements elaborated in Section 2. The requirements established in the research can be used as the benchmark for evaluating the functional utility of the Marama meta-tools. The generalised event abstraction framework incorporates compositional primitives as event handling building blocks and allows composition relationships between them. The framework contains reusable designs to allow users to initialise their system and specify customised event types, event generators, event receivers and event handling building blocks to enhance the extensibility and flexibility of the framework. The framework supports tool integration via a canonical data/event model extension and consistent user interfaces. Graphical notations are offered in the style of the three metaphoric exemplars, but are of a common unified representation: Rectangles represent data, Circles represent constraints, and Connections represent relationships. Multiple views can be easily navigated from one to another. Textual notations are also permitted so that users can escape to conventional code when specifying complex custom behaviours such as code generation. Developing prototypes using Marama meta-tools takes

considerably less time than implementing them using a programming language from scratch. The behavioural models generate Java code which is executed as efficient as code implementations.

- A large number of graduate-level student end users (novice short-term research task-oriented users) were involved in an extensive usability study. In the experiments, 122 participants constructed a domain specific visual language tool of their choice, but with a minimum set of tool features that had to be included in their tool, and were then surveyed. The participants were allowed to work either individually, in pairs, or in a team of 3-5. The aim of the experiment was to provide a substantial, realistic tool development situation and obtain qualitative information on user perceptions of the toolset and quantitative task completion data (whether the minimum feature set was in fact implemented). The experiments evaluated whether end users found the Marama meta-tools easy and effective for generating their chosen domain specific visual language tool. We aimed to use the end users' feedback to improve the Marama meta-tools, and significant enhancement was undertaken after the experiments. The task completion data is positive showing that tools with realistic level of complexity (usable tools with both static and dynamic features) can be designed and constructed using the Marama meta-tools in a short period of time (three weeks working alongside other commitments). General weaknesses emphasised in the survey included: the steep learning curve of the Marama meta-tools; the lack of API documentation (users need to have access to API documentation for very complex event processing) and comprehensive user manual; the stability and the ineffective error handling in the prototype; the limited number of reusable building blocks for behavioural specifications; and the difficulty of defining complex formulae due to unfamiliarity with OCL.
- A smaller number of developers (experienced long-term research goal-oriented users) in our research team, who used Marama meta-tools to develop more substantial applications, provided qualitative feedback in the form of experience reports. The advanced applications being developed or integrated with Marama meta-tools include a generic mapping tool, a health care visual modelling environment, a business process integration tool, an architecture modelling/mapping tool, and a design critiquing system. These qualitative feedback reports were used to assess whether our perceptions of the Marama meta-tools needed to be altered for more experienced user groups, and whether additional requirements were needed (e.g. more complex back end integration requirements). While some event handling building blocks can be used effectively to compose event-based behaviour specifications, all the expert developers needed to escape to code (i.e. use the original custom code writing approach) to define complex backend code generation and user interface extensions (particularly for complex layouts). This indicates to us that the Marama meta-tools need to be further generalised from more examples so that it can provide support for a wider-range of event-based system specifications.

Substantial efforts have been taken to improve the Marama meta-tools based on these evaluation results. The Marama meta-tools have been made more stable and more resistant to incorrect specifications so that a generated DSVL tool can be error-free for use.

A set of JUnit-based test suites are under development. They will be used to perform automatic testing on the Marama meta-tools, particularly the behaviour model. This will remove much of the effort of the developers in undertaking white box, black box, unit, integration and system testing, and allow more focus to be placed on end user usability studies.

Our evaluation approach has demonstrated its effectiveness in eliciting weaknesses of a software prototype, so we are reusing the approach to conduct iterative evaluations on the Marama meta-tools. However, from the previous evaluation results, we found that the major barrier for users to effectively use the Marama meta-tools was the initial steep learning curve. To remove this barrier, we plan to provide the end users with more interactive, story-telling examples in a video-format tutorial so that they learn the Marama meta-tools in a more constructive way. We plan to follow the set of guidelines for developing such videos suggested by Plaisant and Shneiderman [17].

6. CONCLUSIONS

Our research has focussed on providing visual specification and runtime visualisation support for the design and construction of complex event-based systems. We have integrated three event handling specification languages. They are in the domain of web services and business process composition systems, graphical user interfaces (GUI) systems, and constraint-based meta-modelling systems respectively. A synergy of these languages and their generalisation in the Marama meta-tools environment provide wider-ranging support for event-based system design and construction. The event handling abstraction framework provided with Marama meta-tools contains a canonical meta-model representation (generic model) of event handling specifications that enables multiple behavioural paradigms to be easily integrated into the framework. ViTABaL-WS, Kaitiaki and MaramaTatau provide visual languages and tools for event handling specification. ViTABaL-WS is used for high-level conceptual modelling of event propagations among Marama components; Kaitiaki is used for intermediate level design of event propagations among a set of user or library defined queries, filters and actions; MaramaTatau is used for implementation level specification of model and view value dependencies and constraints. The three distinctive behavioural modelling views are wired together by their underlying model. The generic event handling model generates Marama XML, EMF notifications and Java event handlers to be interpreted by the Marama framework for dynamic queries and updates of models and views.

The Marama meta-toolset along with its event handling abstraction framework is still at the prototype stage. We aim to continually develop it to be a robust open source software system to be freely used by interested researchers and organisations. A range of possible future work directions exist developing from such a platform:

- More complete checking of behaviour models could catch errors in the specification before code generation and realisation.

- Users must currently manually layout both the structural and behavioural model views. Automatic layout may be useful to improve a user's ability to show/hide/collapse parts of a specification to manage size and complexity. Some layout specifications have been implemented in a selection of Marama related projects, and it is worth generalising a common layout representation in the same way that we have generalised the event handling abstraction framework.
- Programming by example extensions would be useful in every view of the Marama meta-tools to allow users to explore and instantiate from existing models.
- The visual debugger could be further enhanced with "watch" controls for more efficient tracing and analysis of interested events and their handling behaviours.

The event handling abstraction framework is to be evolved by abstracting from more domain-specific examples. The abstraction model can be specified in the MaramaTorua [12] mapping tool to facilitate generation to a wide range of implementations for interpretation. To allow one specification to generate others with corresponding implementation classes, a set of mapping schemas can be defined in MaramaTorua to provide interchanging mechanisms between distinct domain specifications. MaramaTorua is integrated with the Marama meta-tools and its generated translators can be used directly within new Marama tools to support model integration, translation, and code and script generation.

Many other Marama extensions are being developed. These include a distributed environment with thin client user interfaces and web service back-end, collaborative support for concurrent team work, sketch-based user interfaces and automatic translations to formal Marama model and views. Once these extensions are fully developed, we will integrate them into the Marama meta-tools, generalising further the event abstraction model, and thus making the framework more full-fledged.

7. REFERENCES

[1] 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.

[2] 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. p. 254-259.

[3] Dillon, A., *Usability evaluation*. W. Karwowski (ed.), Encyclopedia of Human Factors and Ergonomics, 2001.

[4] Green, T.R.G. and M. Petre, *Usability analysis of visual programming environments: a 'cognitive dimensions' framework*. J. Visual Languages and Computing, 1996. **7**: p. 131-174.

[5] Grundy, J.C. and J.G. Hosking, *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.

[6] Grundy, J.C. and J.G. Hosking, *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.

[7] Grundy, J.C., J.G. Hosking, and W.B. Mugridge, *Supporting flexible consistency management via discrete change description propagation*. Software - Practice and Experience, 1996. **26**(9): p. 1053 - 1083.

[8] Grundy, J.C., J.G. Hosking, and W.B. Mugridge, *Towards a unified event-based software architecture*, in *the SIGSOFT'96 Workshops, 1996 International Software Architecture Workshop*. 1996, ACM Press: San Francisco. p. 121-125.

[9] Grundy, J.C., J.G. Hosking, and W.B. Mugridge, *Visualising Event-based Software Systems: Issues and Experiences*, in *SoftVis97*. 1997: Adelaide, Australia.

[10] 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.

[11] Hartson, H.R., T.S. Andre, and R.C. Williges, *Criteria for evaluating usability evaluation methods*. International Journal of Human-Computer Interaction, 2003. **15**(1): p. 145-181.

[12] Huh, J., et al., *Integrated data mapping for a software meta-tool*, in *the 20th Australian Software Engineering Conference*. 2007: Gold Coast, Queensland, Australia

[13] Liu, N., J.C. Grundy, and J.G. Hosking, *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.

[14] Liu, N., J.C. Grundy, and J.G. Hosking, *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.

[15] Liu, N., J.G. Hosking, and J.C. Grundy, *MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism*, in *the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing*. 2007: Coeur d'Alène, Idaho, USA.

[16] Paschke, A., *ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language*, in *Int. Conf. on Rules and Rule Markup Languages for the Semantic Web (RuleML'06)*. 2006: Athens, Georgia, USA.

[17] Plaisant, C. and B. Shneiderman, *Show me! Guidelines for producing recorded demonstrations*, in *the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. 2005: Dallas, USA. p. 171-178.

[18] Roberts, D. and R. Johnson, *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, in *the Third Conference on Pattern Languages and Programming*. 1996, Addison-Wesley.

[19] Zhu, N., et al., *Pounamu: a meta-tool for exploratory domain-specific visual language tool development*. Journal of Systems and Software, 2007. **80** (8).