(c) IEEE 2004. In Proceedings of the 2004 IEEE Int Conf on Automated Software Engineering.
Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool

Yuhong Cai[1], John Grundy[1,2] and John Hosking[1]
Department of Computer Science[1] and Department of Electrical and Computer Engineering[2]
University of Auckland, Private Bag 92019, Auckland, New Zealand
{rainbow,john-g,john}@cs.auckland.ac.nz

Abstract

We report on our experiences developing a performance test-bed generator for industrial usage by extending an open-source UML CASE tool. This tool generates client and server code, database configuration and deployment scripts from a high-level software architecture description. It automates the code generation, compilation, deployment and performance metric result collection processes. We identify a range of problems that arose from our previous research on performance test-bed generation that needed to be addressed to scale this automated software engineering technique. We describe a range of approaches we used to solve these problems in our new tool. We then report on industrial deployment and evaluation of our new tool and discuss the effectiveness of these solutions.

Keywords: experience report, architecture analysis, software performance testing, software tool extension

1. Introduction

Non-functional software performance requirements, such as response time and transaction throughput, are in general very challenging to meet. For example, estimating likely target application performance from a high-level software architecture design is a very hard problem [1, 4, 7, 19]. A range of approaches have been attempted, including rapid prototyping, existing system profiling, architecture modelling and simulation, and performance test-bed generation [11, 7, 5, 1, 12, 8].

In our previous work we developed SoftArch/MTE [8], a performance test bed generator that generates client and server test-bed code and deployment scripts from a high-level software architecture design model. The tool automatically generates, compiles and deploys the test bed code, runs the specified performance tests on the code, and reports the performance results to a software architect. Initial experiences using this tool and approach were very promising. However, when we tried to apply our prototype tool to industrial case studies with a view to commercialising this automated software engineering technique, we encountered a number of problems. These included: use of a non-standard architecture modelling notation and prototype architecture design tool; non-standard representation of the architecture model; limitations on the expressiveness and maintainability of the code generators and the simplistic code compilation and deployment tools we had built; and unsuitability of the performance data management and visualisation support.

The focus of this paper is to present a number of approaches that we have used to solve these challenges and report on our experiences with these techniques. These have included: extending an open-source CASE tool, ArgoUML, to provide UML-like architecture modelling and XMI-derived model representation capabilities; restructuring and enhancement of the XSLT-based code generators employed; use of Ant, an open-source build tool, to manage complex automatic process of code generation, compilation and deployment dependencies; use of SFTP, a widely supported file transfer protocol to manage code deployment and performance result capture; an MS Access database for performance test management; and result visualisation plug-ins for the extended Argo CASE tool. The resultant industrial-strength performance test-bed generation tool is called Argo/MTE.

We first motivate this work by describing approaches to performance estimation and in the process describe and critique our original SoftArch/MTE performance test-bed generation tool, identifying its strengths and weaknesses for industrial usage. We then describe the approaches we used to build our new Argo/MTE tool in order to scale-up the capabilities of SoftArch/MTE. Each of the key approaches we used is motivated, described and illustrated. We then report on the deployment of Argo/MTE on two industrial application projects and reflect on how successful our techniques were. Based on this we discuss applicability of techniques we used for other automated software engineering systems with similar characteristics.

2. Motivation

Many approaches have been used for performance estimation. These include benchmarking [1, 7] which uses reference architectures and load-testing of simple implementations. Relative performances of the differing technologies used in the implementations are compared. While accurate measures for the particular benchmark

application are obtained, these only provide a rough guide for related applications [7]. Rapid prototyping [11] involves developing partial applications for performance-critical parts of a system e.g. network-centric and database-intensive. This can require significant development effort particularly if architecture evolution requires prototypes to be modified and tests repeated. Analysis of deployed systems [13, 15, 17] involves studying performance patterns and trying to extrapolate these onto new systems with similar architectures. This approach is limited to providing bench-mark style guidelines only for architects. Simulation approaches model distributed applications and simulate performance with over-head estimates based on architecture [1, 12] or middleware [12, 19] choices. Accuracy varies widely and it is difficult to obtain performance models for 3rd party applications such as databases. Performance tuning tools [14] are post-implementation approaches to improving architecture performance, but are limited by the architecture design adopted as to the effect they can have. SoftArch/MTE is a performance estimation tool we have developed which uses a different approach: test bed generation [8]. This is an automated form of the rapid prototyping approach that others have recently begun experimenting with [4].

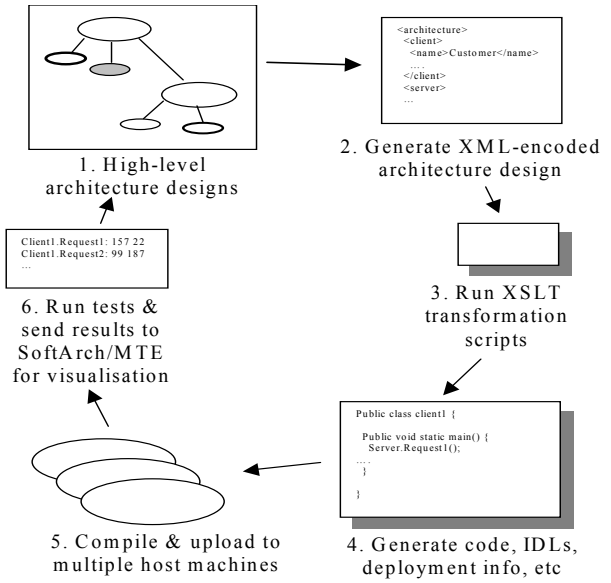


Figure 1. Outline of the SoftArch/MTE architecture performance analysis process (from [8]).

Figure 1 illustrates how SoftArch/MTE is used to gain an understanding of the likely performance of a software architecture design. The architect constructs one or more high-level architecture designs, specifying clients, servers, remote server objects and database tables, client-server, server-server, client/server-database requests and server services, and various kinds of architectural connectors. They specify properties such as client, server and database host; number and frequency of requests (e.g. call remote method 1000 times; call repeatedly without delay; call every 2 seconds; etc); database table and request complexity (e.g. one or multiple row select; 10 row update; one row insert/delete etc); middleware protocol (e.g. CORBA, web services message, RMI etc); and so on. SoftArch/MTE generates an XML encoding of the architecture model (2) which is then passed through a number of XSLT-based (XML style sheet transformations) scripts (3). These scripts generate, without any user intervention, Java, C#, etc code, JSP and ASP web server components, CORBA, COM and WSDL 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 runnable performance test-bed that when executed generates real client-server-database interactions and captures performance profile information (5). Compiled code is up-loaded to client, server and database hosts. The generated programs are started on all hosts and when signalled clients begin execution i.e. send requests to their servers. Code annotations and/or 3rd party profiling tools capture performance measures and these are returned to SoftArch/MTE. Diagram annotations, property sheets and Excel worksheets are used to show performance measures.

We have used SoftArch/MTE to evaluate the performance of a range of software architectures. The performance results obtained have been validated against both hand-implemented realisations of the architectures tested and hand-implemented performance test-beds [9]. However, these experiences and our attempts to use the tool on several other industrial projects revealed major deficiencies in the realisation of this automated software engineering technique. These included:

- *Non-standard design tool and modelling notation*. SoftArch is an experimental proof-of-concept architecture modelling tool which uses a non-standard visual architecture modelling language. While it proved suitable for experimenting with and evaluating the performance test-bed generation concept, it has poor usability, limited integration with other CASE tools which causes a steep learning curve for tool users.
- *Proprietary XML architecture model format*: SoftArch saves model designs in an ad-hoc XML model format we developed only for our own experimental work. This makes exchanging the design with other tools difficult and excessively couples the architecture design tool and performance test bed code generators.
- *Unmaintainable code generators*. Our SoftArch/MTE code generators demonstrated that XSLT provides a good implementation platform. However, the original prototype XSLT scripts proved hard to maintain, difficult to debug and impossible to reuse effectively.

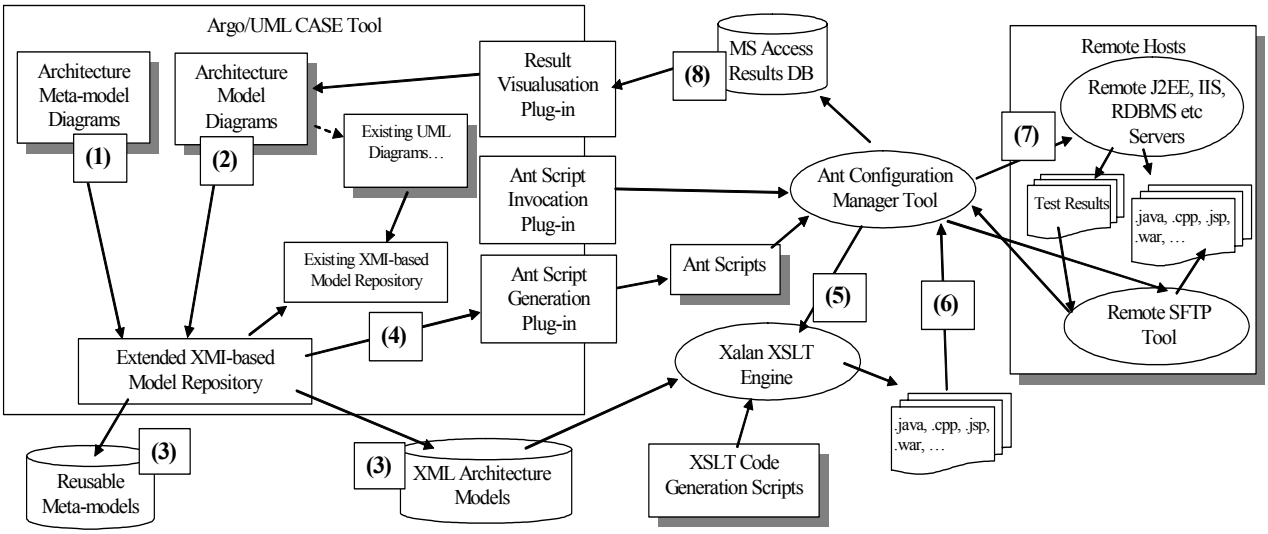


Figure 2. Overview of Argo/MTE architecture.

- *Tool-driven code generation, compilation and deployment.* In SoftArch/MTE we had the tool itself co-ordinate the code generation, compilation, deployment and also result capture and visualisation (steps 3-6 in Figure 1). When adding new target code and scripts e.g. JSP and ASP web server components, web services WSDL descriptions and deployment scripts, and trying to deploy the tool on secure networks, this approach proved very hard to maintain and tailor to different tool deployment environments.
- *Proprietary code deployment tool.* We built our own Java deployment tool client and server for SoftArch/MTE. This worked well but again proved to be too difficult to adapt to different tool deployment environments, hard to maintain and lacked fundamental code deployment and test control facilities.
- *Result capture and visualisation.* We added bespoke code to SoftArch to visualise performance results in various ways. This proved difficult to extend to capture other kinds of results and visualisation facilities provided were poorly integrated with other facilities. Managing and comparing multiple architecture design results was particularly poorly supported.

Scaling up automated software engineering techniques for use on large industrial problems has long been of interest [2]. Techniques adopted include integration with Commercial Off-The-Shelf (COTS) tools [6], developing more robust implementations for large-scale problem domains, and adopting widely used standards, enabling reuse of industrial-strength tools developed by others [16].

3. Overview of Argo/MTE

To solve the problems we encountered with SoftArch/MTE and produce an "industrial strength" performance test-bed generation tool we developed a new tool, Argo/MTE. ArgoUML is a widely-used open source CASE tool [20, 21] incorporating other open source packages such as novosoft UML, java XML and java GEF. It provides well organized source files and documentation. We extended ArgoUML to incorporate our performance test-bed support in an integrated UML-based CASE tool.

Argo/MTE is an integrated development environment for software architecture design and architecture performance evaluation. Figure 2 provides an overview of the Argo/MTE architecture and its usage. Multiple Argo/MTE domain-specific meta-models can be defined using a new tool we added, each providing a different set of architecture modelling abstractions and code generators e.g. for web-based or real-time systems, etc (1). These meta-model abstractions are stored using an extended form of ArgoUML's XML Meta-data Interchange (XMI)-based XML model representation format within the ArgoUML environment. Argo/MTE allows tool users to draw, modify, refine, and revise software architecture designs, again using a new architecture modelling tool we added to ArgoUML (2). Architecture models are developed using one or more meta-models and multiple design views. Each Argo/MTE design encodes enough data to generate a test bed (normally a distributed software system) for a given level of abstraction. The test bed not only implements fundamental functional requirements of the intended system, but also carries performance evaluation information. Meta-models and architecture models are saved in an XML file format (3). A set of Ant configuration management tool scripts are generated to perform code-generation, compilation, deployment, test initiation and results capture (4). The XML-encoded software architecture model is transformed into a range of

files and scripts (5). A set of XSLT scripts and the Xalan XSLT engine perform this work under Ant script control.

The generated test-bed code is compiled and deployed to multiple host machines (6). Performance tests are then run producing text files capturing the performance profiling results (7). The results are downloaded and captured in an MS Access database, producing an archive of architecture model/performance results over time. The result database is queried and performance results for a single or multiple performance test runs visualised using various graphs and architecture model annotations (8).

In the following we examine in more detail the architectural and design decisions made when developing Argo/MTE to solve the scalability of SoftArch/MTE. These include: the architecture modelling and meta-modelling capabilities; architecture model representation and management; co-ordination of code generation, compilation and deployment; and performance test execution, data management and visualisation.

4. Architecture Modelling

User evaluation of SoftArch/MTE identified several enhancements needed for architecture modelling support [9]. These included software architects' desire for a more conventional UML-based modelling language, integration of our technique into a "standard" UML-based CASE tool, and ability to modify and extend the architectural design tool's meta-model types. To achieve this we chose to extend ArgoUML with SoftArch/MTE-style capabilities. We chose ArgoUML instead of other UML CASE tools for several reasons: it is open source; well-structured and extendable at both diagramming and model levels; uses common data representation standards such as XMI; and its cognitive support could be used to provide architecture design process support [20]. We made three fundamental extensions to ArgoUML to support architecture modelling:

- extending ArgoUML's data structures to support architecture-specific meta-model types and model instances using these domain-specific types
- adding a visual meta-model type specification tool
- adding a visual architecture design tool

We designed modelling elements at UML meta-model level to support Argo/MTE-style architecture modelling. We extended ArgoUML's existing UML meta-model data representation to capture these software architecture designs. Figure 3 shows our design of modelling elements at UML meta-model level. Modelling capabilities added include representing host machines, nodes (processes), operations, attributes and various parameters. We tried to ensure this design followed logically from existing Argo meta-model types. An Argo/MTE domain-specific meta-model provides a modelling framework for a particular domain, which specifies available modelling types, properties and their relationships available to define an architecture model's structure and semantics.

Each architecture design model in Argo/MTE must use one or more previously defined meta-models. Meta-models can be defined for e.g. "web-based applications", "real-time applications", "mobile device applications", etc.

Each component in an architecture design model is typed by a meta-type (a meta-modelling element) which specifies allowable properties and relationships to other elements. Each such meta-type defines sets of architectural and testing parameters. Architectural parameters define architecture design structural data while testing parameters provide performance code generation-related data. A good meta-model is essential to ensure an architecture design has adequate information with low redundancy.

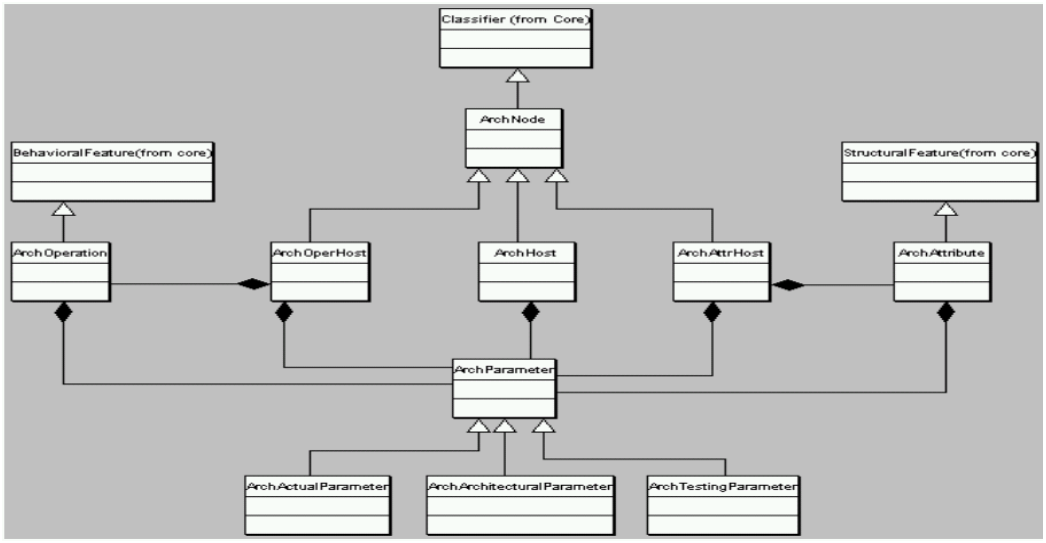


Figure 3. ArgoUML XMI meta-model extension examples.

We developed a meta-model specification tool for Argo/MTE allowing architects to build and refine their own meta-models. This was developed as a specialisation of the ArgoUML class diagramming tool. Figure 5 (a) shows a sample e-commerce meta-model for CORBA-based information systems modelling. This meta-model includes abstractions for client, database and, application servers, remote objects, and other architecture modelling types. Argo/MTE uses the ArgoUML view layout menu and tool bars (1,2), model element tree view (3), diagram editing pane (4), and tabbed property sheet pane (5).

To build an Argo/MTE meta-model a target application domain is analysed and abstract modelling element types devised for that domain. A number of architectural elements from the CORBA-based information systems domain are shown in Figure 5 (b). Element attributes with type "AP" are ArchArchitecturalParameters (generally structural) properties. Ones marked "TP" are ArchTesting Parameters used for performance test-bed code generation.

Using a similar approach we developed an architecture modelling tool by specialising the class diagramming and collaboration diagramming tools from ArgoUML. This approach reduced development time and provided architects with design tools similar to the look and feel of the ArgoUML toolset. Part of a complex, distributed micro-payment system architecture, NetPay [3], is shown in Figure 4© to illustrate this tool. The architecture modelling notation we developed extends the UML class and collaboration diagram appearance and layout. We used a class icon-like representation of architecture abstractions rather than UML-style deployment diagram shapes as we found the latter cumbersome and inflexible.

An Argo/MTE model comprises elements (rectangles), element requests and services (labels), associations (solid black lines), message interactions (blue lines and highlights), hosting associations (dashed lines), and refinements (solid or dashed black line with one end point). Stereotypes indicate meta-model type instantiation. Each element has a property set derived from its meta-model type. The architecture in Figure 4(c) comprises a customer PC-hosted browser and payment client ("E-wallet") (1), a broker (2), and several vendor sites (3). The vendor has a multi-tier architecture: the client browser accesses web pages (4), which access application server components via CORBA (5), and a database (6). Each abstraction links to other abstractions via relationships. Properties/parameters for the <<Client>>Reader component are shown below. Architectural parameters support architecture modelling e.g. types and relationships. Testing parameters support performance code generation, including number of client threads, whether to record timing information, middleware threading and transaction models and configuration parameters such as number of times to repeat requests, pauses between requests etc.

Multiple architecture and meta-model type views are supported for complex specifications. Figure 5 shows three views of the NetPay system. Collaboration relationships between client requests and server services (1) visualise/ specify message-passing relationships between elements. (2) shows just the message passing relationships between elements. Refinement of higher-level abstractions is shown in (3), where CustomerRegistrationPage service "register Customer()" is refined to constituent operations (each realised by business logic and database operations). We further developed a concept of architectural model refinement from SoftArch/MTE in Argo/MTE. This allows an architect to specify refinements of high-level concepts e.g. "RemoteServer" to smaller abstractions e.g. to "CustomerManger", "ContentManager" and "PaymentManager". This concept was not supported in ArgoUML and required the addition of cross-diagram and inter-meta-model element refinement relationships. We also allow architects to refine architecture elements from OO analysis classes, and refine them to OO design classes. This provides an integrated traceability support mechanism within the architecture design tools.

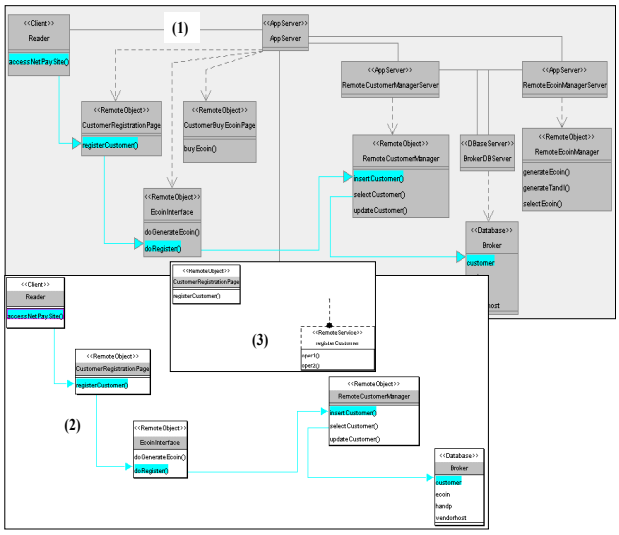
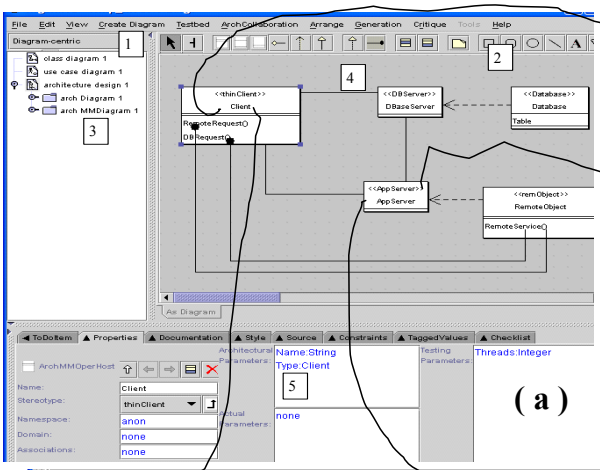


Figure 4. Multiple views and refinement examples.

5. Test Code Generation and Deployment

In our SoftArch/MTE prototype tool we developed a proprietary XML format to represent and store SoftArch designs. This was then processed by XSLT scripts to generate target code files, batch scripts, deployment descriptors, IDL files etc. This worked well for relatively small example systems but proved inefficient and caused XSLT script implementation and processing problems when modelling larger systems. It also has the fundamental problem that only our own tool could ever generate it.

We wanted to represent the architecture models in a more standardised format, and our final aim is to eventually make Argo/MTE model data exchangeable with other XMI-supporting CASE tools.



Element Type	Main Attributes	Property Description
Client : ArchOperHost	ClientType (AP, TP) Threads(TP)	Type of a client e.g. browser, CORBA client. Number of con-current clients run for tests
RemoteRequest : ArchOperation	RemoteServer (AP, TP) RemoteObject(AP, TP) RemoteMethod(AP, TP) RecordTime(TP) TimesToCall(TP) PauseBetweenCalls(TP)	Name of remote server to call The name of remote object The name of remote service Record time for this? Repetitions Pause duration between calls
AppServer : ArchHost	Name (AP, TP) RemoteObjects(AP, TP) Type (AP, TP)	Name of server Objects this server hosts Type of the application server, e.g. CORBA, RMI, J2EE…
RemoteObject : ArchOperHost	Name, Type (AP, TP) Threading (TP)	Name+type info e.g. EJB, JSP Thread information e.g. #
RemoteService : ArchOperation	Name, Type (TP, AP) Parameters (TP) ConcurrencyCtl (TP) RecordTime (TP)	Name+type info e.g. method Arguments for operation Concurrency management code info Record time for this and how
DBRequest	…	
DBTable	…	

(b)

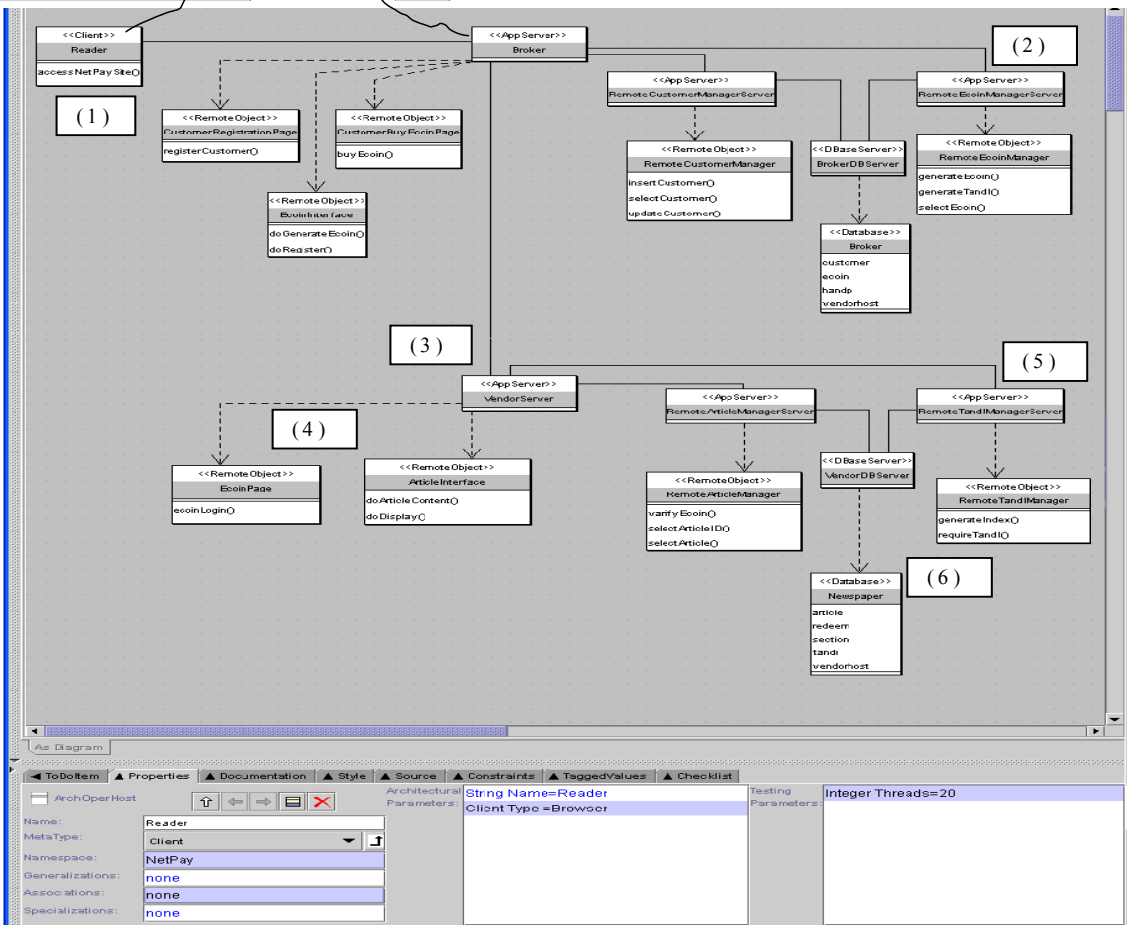


Figure 5 (a). Example of a domain-specific meta-model; (b) Examples of meta-model types and attributes; (c) Example of an Argo/MTE architecture model for complex micro-payment system architecture.

For now we chose to extend ArgoUML's XMI-based UML representational format to capture model structure and code generation parameters. Currently, Argo/MTE model files consist of modelling elements represented using the meta-types shown in Figure 3. We extended the ArgoUML save file module to store and retrieve our extended architectural design models.

We chose to continue using XSLT scripts to implement our Argo/MTE code generators as they proved to be flexible, powerful and more easily maintained and extended than Java or C++-implemented code generators we had used in other CASE tool work. However, we needed to modify the SoftArch/MTE scripts to use our extended model format and also had to make the code generators more efficient and generate larger but more easily managed target code.

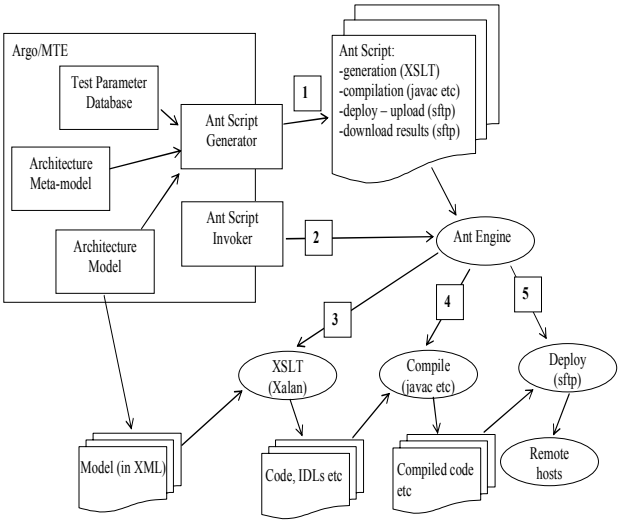


Figure 6. Argo/MTE code generation, compilation and deployment via generated Ant scripts.

In SoftArch/MTE a set of code generation XSLT scripts were run over multiple XML save files generated by the tool. The scripts run and input files to give them were determined by examining the architecture model. Each script was responsible for organising its own output file(s). We discovered when adding code generation scripts for JSP and ASP web server components, J2EE Enterprise Java Bean code and deployment descriptors, and .NET component code and WSDL interface definition files that this approach didn't scale well. It often required hand modification of SoftArch/MTE's code generator co-ordination code and it was difficult to order invocation of compilation, deployment and test execution scripts.

To solve these problems we have structured our Argo/MTE code generator to have scripts associated with meta-model types. In addition, we have developed a repository of code generation parameters which are used to determine which scripts require running on a model, their order and templates for generating a configuration management tool script to handle the dependencies between code generation scripts and moving and packaging their output files. This approach does not require any code modification to Argo/MTE when new scripts are added, just maintenance of the test parameter repository.

In addition to hard-coding the code generation, SoftArch/MTE hard-coded compilation and deployment of generated code, batch files, IDL files etc. This also proved non scalable. In Argo/MTE we replaced this with an Ant script generator component. This uses test parameters, meta-model types and model data to generate an Ant script that runs: XSLT code generation script processing; generated code compilation, including source code and IDL files; compiled code, IDL file and deployment descriptor uploading to remote hosts; and performance result downloading. Ant manages complex dependencies between parts of a large set of test-bed programmes at each stage of compilation, deployment and test execution. Figure 6 shows how Argo/MTE provides scalable generation, compilation and deployment processes.

In addition to Ant script generation, Argo/MTE also used a 3rd party secure FTP (SFTP) program for deployment and result upload, replacing another hard coded component of Softarch/MTE. We used generated Ant script macros to drive SFTP to both deploy compiled code and download test result files from remote client and server host machines. This provides us with a standardised, widely available file up and downloading facility with an extensible, scripting-based control language.

6. Test Data Management and Visualisation

In SoftArch/MTE deployed client and server programs are initialised, and clients told to execute via our custom deployment tool, and performance results were stored in text files. These were downloaded to the SoftArch/MTE host, parsed and data extracted to visualise test results. Recording of results was by archiving result files and by custom code added to SoftArch/MTE to annotate diagrams and drive simple Excel spreadsheet charting functions.

We found this approach unsatisfactory when adding new code generators for thin-client JSP and ASP server components. We had to generate "pseudo-browser" clients which provided results of limited accuracy. Controlling test runs via the custom deployment tool required continual update of this application. Visualisation of results in SoftArch/MTE was not well-integrated with modelling views and the Excel charting cumbersome. The capture and comparison of test runs for different architecture models was particularly poorly supported.

Figure 7 illustrates the Argo/MTE test deployment, execution and results management process. Argo/MTE instructs Ant to upload and initialise generated test client and server programs, scripts, IDLs and database scripts (1). The generated Ant build script is run with "deployment" parameter, resulting in multiple file uploads to remote hosts using a local SFTP client and remote SFTP servers (2). Each remote host has another generated Ant script uploaded as part of this deployment process.

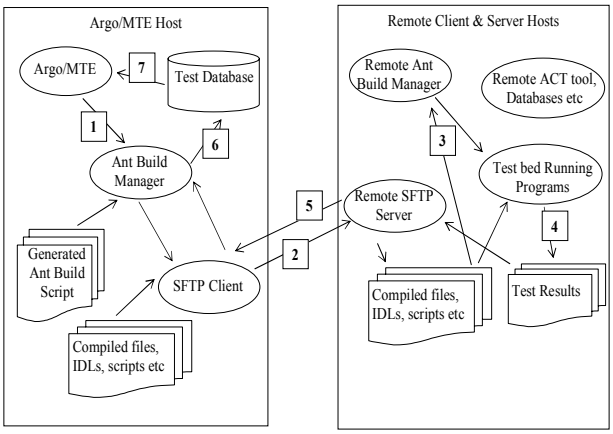


Figure 7. Argo/MTE test execution & results capture.

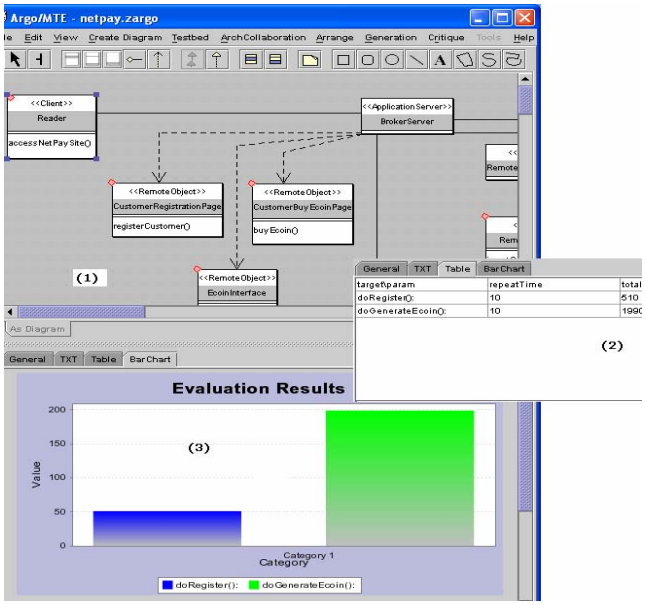


Figure 8. Example of result visualisation.

This is used by a remotely-deployed Ant build engine running on each host to initialise deployed programs and configuration scripts (3). This remote Ant engine script is also used to synchronise the start of multiple client programs. For thin-client test-beds (e.g. JSP or ASP web server components), we use the Microsoft Application Centre Test (ACT) automated test execution tool which simulates multiple browser requests to these JSPs and ASPs, giving realistic browser performance measures. An ACT script is generated for each thin-client from the Argo/MTE model and the remote Ant engine script initialises and instructs ACT to run these scripted tests. Results from performance profiling annotations to code or ACT output files are captured as text files (4). The Argo/MTE Ant engine downloads these results via SFTP (5). It updates a test database, inserting performance results for each model item grouping them by test run (6). The result visualisation component extracts test run results to visualise (7).

We added a result visualisation component to Argo/MTE which queries the test database and displays test run results as graphs or annotations to architecture model diagrams. This is shown in Figure 8; architecture components with performance results available are annotated with a small box at the top left corner. These results can then be examined in tabular or graph form. We decided after user evaluation of SoftArch/MTE to more closely integrate the result visualisation support into Argo/MTE architecture model diagrams. Users found our previous approach of using an Excel charting macros and generating new SoftArch/MTE annotated views to be difficult to use. Our new approach of a database to capture multiple test run results allows us to much more easily visualise multiple test runs e.g. comparing the relative performance of different architecture design decisions.

7. Discussion

7.1 Industrial Deployment of Argo/MTE

We have carried out two detailed industrial case studies using Argo/MTE, one on a complex micro-payment architecture [3] and one on an enterprise integration system architecture [10]. In both cases we had example architectures and fully-implemented systems. We had also carried out performance analysis experiments with the hand implemented systems in previous projects so we could model and compare the Argo/MTE-produced results with these actual performance results. In addition, with Argo/MTE we could model alternative architecture design scenarios and obtain performance results to compare to the results from other architectures. We used these case studies as they both provided us with complex architectures to model and performance test, and we were familiar enough with the problem domains that we could analyse and interpret the results in depth. For both case studies we had to add new meta-model abstractions, code generators and build compilation script templates. Our previous SoftArch/MTE prototype proved very difficult to adapt and use on these case studies whereas we were able to successfully use Argo/MTE.

Our experiences with these large case studies were generally very positive. We were able to build successive levels of abstract and detailed architecture models with Argo/MTE, which provided a robust environment and effective set of architecture modelling facilities. Extensions to the meta-model, code generation scripts and build and deployment configuration templates were all much more straightforward than with SoftArch/MTE. We were able to generate code for and compile very complex test beds. Both systems had several servers and databases, with numerous remote operations per server, even with greatly simplified architectural models. We required a

range of test bed clients, many run con-currently, to request a mix of different server operations and gather overall performance metrics from the generated server programs. Overall, the performance results we obtained from Argo/MTE's generated test beds were reasonably close to those obtained when running the exact same clients against the real implemented system servers. We discovered some implementation errors in the real micro-payment system example, badly affecting its performance. These were highlighted by obtaining Argo/MTE test bed results wildly different from those obtained from the real system. Correction of these programming errors resulted in much closer performance result correlation.

Several limitations were encountered when using Argo/MTE on these case studies. The first related to the inability of the architect to specify ranges of values rather than absolute numbers of clients, client requests, etc. Other included the effort needed to add and test new code compilation and build support, and the limited result visualisation in the tool at present.

7.2 Advantages and Disadvantages of Argo/MTE

Argo/MTE provides integrated modelling support as an addition to a UML-based CASE tool, ArgoUML. This was found to be a much more appealing and effective environment than our previous, stand-alone SoftArch/MTE tool. The use of a set of UML-like modelling abstractions provides an architecture design notation closer to designer's other modelling languages within this tool. The use of an XMI-like model representation format and extensible architecture meta-models increases the chance of model data exchange. Restructuring our XSLT-based code generators and using the Ant build manager tool to control code generation, compilation and deployment greatly eased this process. It is now far easier to add new meta-model abstractions to represent new target middleware or database technologies, add new code generators, compilers and deployment scripts, and generate Ant build scripts to control this process. The use of third-party tools to co-ordinate the test bed generation and execution process (Ant), deployment (SFTP), and web-based client tests (ACT) has proved much more scalable and flexible than our previous ad-hoc applications to perform these tasks. We found this particularly so when heterogeneous architectures incorporating several technologies had to be generated and deployed.

Several issues need to be further addressed with Argo/MTE. This includes process support to guide the use of the tool. The design of an architecture, performance test-bed generation, and evolution of the architecture design using test results has proved to be a complex process requiring considerable domain knowledge. Support for this may include use of ArgoUML's critics to provide pro-active guidance to architects for the performance testing [20]. We have used an extended, XMI-like format to represent Argo/MTE model data, but this is not standard. As the UML is enhanced to incorporate more architecture representation facilities, our approach ideally will need to be adapted to use these, from an XMI-based encoding of the model. Similarly, if UML architecture modelling approaches become standardised, we would like to use these instead of our architecture modelling enhancements. We found writing new XSLT code generators and Ant build script templates one of the most time-consuming aspects of adding new target technology support to Argo/MTE. Adding an IDE for these facilities to the tool would make it much easier for architects to extend its supported target technologies.

Storage of test results in an MS Access database proved a major improvement. However, while SoftArch/MTE's use of Excel charts to visualise results proved to be unpopular with end users, our Argo/MTE approach of an Argo extension to display results is also limited. Extending its support to display results in different ways and to carry out different analysis of results requires extra Java coding. An enhancement would be to use MS Access reports to generate these visualisations but to incorporate the invocation and display of report results within the Argo/MTE using OLE or similar component integration.

7.3 Generalisation of Our Experiences

While it is difficult to generalise from one example, we feel that there are a number of lessons from our experiences in scaling up our test bed generator technique that may be applicable to scaling of other automated software engineering techniques.

The first of these is to where possible concentrate on using commonly used COTS (or Open Source OTS) tools to implement parts of the application rather than developing bespoke code. Such tools have typically been developed with efficiency, both in terms of performance and representational economy in their target domain, in mind [6]. They have also been through the quality control processes needed to support industrial usage. Examples of this in Argo/MTE, which can be considered as lessons in their own right are:

- Use of build tools to manage complex dependencies. This is a specialist domain, fraught with difficulty. Reuse of existing expertise to solve the compilation and deployment problem was an essential component of Argo/MTEs successful development
- Use of scripting languages and engines. Both Ant scripting and XSLT scripts proved excellent approaches for representing complex operations. Scripting notations are tuned to their particular domains of application, hence are terse and relatively easy to develop code generators for.
- Use of databases for management and visualisation of large datasets over time.

The second major lesson is the use of standard representations where possible. While our SoftArch notation was novel and innovative, this proved to be a

handicap for user adoption. Use of the UML meta model for defining our architecture notations proved successful for two reasons. Firstly it provided users with a much more familiar set of notations, extending obviously from UML class and object diagrams. Secondly it meant we could reuse the ArgoUML modelling tools, by specialising them to suit our notation extensions. This saved considerable development effort in developing the modelling tools.

Key future enhancements we plan to explore include: using the Eclipse [18] framework instead of ArgoUML; a fully XMI-compliant XML representation using UML meta-model derived architectural modelling types; and an IDE to assist in developing new XSLT and Ant code generation and built scripts for new target technologies.

8. Summary

Applying our SoftArch/MTE automated performance test-bed generation tool to industrial case studies proved problematic. We found that while this automated software engineering technique was applicable to the case study domains, the proof-of-concept tool had many problems when trying to scale it. We have developed Argo/MTE, where we integrated the test-bed generation approach into an open source, UML-based CASE tool. Extensions of UML modelling notations and data representations of models are used to describe architectures. A number of third-party tools, including XSLT, Ant, SFTP and MS Access, are used to realise the performance test-bed generator support in a much more scalable and flexible way. Using Argo/MTE on two large industrial case studies indicates these approaches have generally been successful in scaling our test-bed generation approach.

References

1. Balsamo, S., Simeoni, M., Bernado, M. Combining Stochastic Process Algebras and Queuing Networks for Software Architecture analysis, *Proc 3rd Intl Wkshp Software & Performance*, 2002, ACM Press.
2. Balzer, B. A 15 year perspective on automatic programming, *IEEE Transactions on Software Engineering*, vol. 11 no. 11, Nov 1985, pp.1257-1268.
3. Dai, X. and Grundy, J.C. Architecture for a Component-based, Plug-in Micro-payment System, Proc 5th Asia-Pacific Web Conference, Sept 27-29 2003, Xi'an, China, LNCS 2642, pp. 251-262.
4. Denaro, G. Polini, A. and Emmerich, W. Early Performance Testing of Distributed Software Applications. In Proceedings of the 4th International Workshop on Software and Performance, Redwood Shores CA, Jan 14-16, 2004, ACM Press, pp. 94-103.
5. ECPerf Performance Benchmarks, August 2002, ecperf.theserverside.com/ecperf.
6. Egyed, A. and Balzer, R. Unfriendly COTS Integration-Instrumentation and Interfaces for Improved Plugability, In *Proceedings of the 2001 IEEE International Conference on Automated Software Engineering*, San Diego, 26-29 Nov 2001, pp. 223-231.
7. Gorton, I. And Liu, A. Evaluating Enterprise Java Bean Technology, In *Proc Software - Methods and Tools*, Wollongong, Australia, Nov 6-9 2000, IEEE.
8. Grundy, J.C., Cai, Y. and Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Descriptions, Proc 2001 IEEE Intl Conf on Automated Software Engineering, San Diego, CA, Nov 26-29 2001.
9. Grundy, J.C., Cai, Y. and Liu, A. SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions, accepted for publication in *Automated Software Engineering*.
10. Grundy, J.C., Bai, J., Blackham, J., Hosking, J.G. and Amor, R. An Architecture for Efficient, Flexible Enterprise System Integration, In *Proceedings of the 2003 International Conference on Internet Computing*, Las Vegas, June 23-26 2003, CSREA Press, pp. 350-356.
11. Hu, L., Gorton, I. A performance prototyping approach to designing concurrent software architectures, In *Proc of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, IEEE, pp. 270 – 276.
12. Juiz, C., Puigjaner, R. Performance modelling of pools in soft real-time design architectures, *Simulation Practice & Theory,* 9, 2002, 215-40.
13. 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.
14. Killelea , P. *Web Performance Tuning*, 2nd Edition, O'Reilly March 2002.
15. McCann, J.A., Manning, K.J. Tool to evaluate performance in distributed heterogeneous processing. *Proc 6th Euromicro Workshop Parallel & Distributed Processing*, IEEE, 1998, 180-185.
16. Nentwich, C., Capra, L. Emmerich, W. and Finkelstein, A. xlinkit: a consistency checking and smart link generation service, *ACM Transactions on Internet Technologies*, vol 2, no. 2, 2002, pp. 151-185.
17. Nimmagadda, S., Liyanaarachchi, C., Gopinath, A., Niehaus, D. and Kaushal, A. Performance patterns: automated scenario based ORB performance evaluation, Proc *5th USENIX Conf on OO Technologies & Systems*, USENIX, 1999, 15-28.
18. Object Technology International, Inc, Eclipse Platform Technical Overview, 2003, http://www.eclipse.org/whitepapers/eclipse-overview.pdf
19. Petriu, D., Amer, H., Majumdar, S., Abdull-Fatah, I. Using analytic models for predicting middleware performance. In *Proc 2nd Intl Workshop on Software and Performance*, ACM 2000, pp.189-94.
20. Robbins, J.E. and Redmiles, D.F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, In *Proc CoSET'99*, Los Angeles, May 1999, pp. 61-70.
21. 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.