Customized Benchmark Generation Using MDA

Liming Zhu[1,2], Yan Liu[1], Ian Gorton[1], Ngoc Bao Bui[3]
[1]*Empirical Software Engineering Program, National ICT Australia Ltd.*
[2]*School of Computer Science and Engineering, University of New South Wales, Australia*
{Liming.Zhu, Jenny.Liu, Ian.Gorton}@nicta.com.au.au
[3]*Faculty of Information Technology, University of Technology Sydney, Australia*
NgocBao.Bui@student.uts.edu.au

Abstract

This paper describes an approach for generating customized benchmark applications from a software architecture description using a Model Driven Architecture (MDA) approach. The benchmark generation and performance data capture tool implementation is based on widely used open source MDA frameworks. The business logic of the benchmark application is modeled in UML and generated by taking advantage of the existing generation "cartridges" so that the current component technology can be exploited in the benchmark. This greatly reduces the effort and expertise needed for benchmarking with complex component technology. We have also extended the MDA framework to model and generate a load testing suite and automatic performance measurement infrastructure. The approach complements current model-based performance prediction and analysis methods by generating the benchmark application from the same application architecture that the performance models are derived from. This provides the potential for tightly integrating runtime performance measurement with model-based prediction either for model validation or improving model prediction accuracy. We illustrate the approach using a case study based on EJB component technology.

1. Introduction

Software component technologies such as Enterprise Java Beans (EJBs) and .NET have proven successful in the construction of enterprise-scale systems [15]. However, it remains a challenging software engineering problem to ensure that an application architecture can meet specified performance requirements.

Various performance analysis models with prediction capabilities have been proposed to evaluate architecture designs during early phases of the application development cycle [8]. Utilizing these models requires two distinct activities be carried out by the application architect. The first requires the development of specific analytical models based on the application design. The second must obtain parameter values for performance model using measurements or simulation. Both these activities require significant additional effort and specific expertise in performance engineering methods. Hence, they are key inhibitors that have prevented performance engineering techniques from achieving wide-spread adoption in practice [8].

With the growing interest in Model Driven Architecture (MDA) [2] technologies, attempts to integrate performance analysis with MDA and UML have been made, aiming to reduce the performance modeling effort required. The OMG's MDA standard defines a way of transforming business domain models into Platform Independent Models (PIM) and then Platform Specific Models (PSM), and eventually to executable code. PSMs can also include a performance analysis model specific to a performance engineering method. Consequently, recent work has attempted model transformation from UML design models to method-specific performance analysis models. A more comprehensive and elaborate theoretical basis has been discussed in [22]. This work has, to a large extent, improved on the earlier manually-oriented activities related to deriving analytical models from application designs.

Parameter values in these performance analysis models also depend on the underlying component framework used to implement the application. One method to obtain and tune these parameters is to run a benchmark application on the component framework. This approach has proven to be useful [11, 12, 17] with component-based technologies. An effective benchmark suite includes a core benchmark application, a load testing suite and performance monitoring utilities.

There are existing industry benchmark standards and suites (e.g. [6]), but these are not broadly suitable for performance modeling for a number of reasons. First, they are mainly designed for server vendors to showcase and improve their products, rather than reflecting a specific application's performance characteristics. Second, these benchmark suites tend to be expensive to acquire and complex to use.

On the other hand, implementing a custom benchmark suite for a component platform from scratch is also costly and tedious. This is largely due to the complexity of modern component containers and the ad hoc ways they adopt for conducting performance measurement. A benchmark implementation usually requires a large amount of container and infrastructure related *plumbing,* even for a relatively simple benchmark design. Interestingly, this characteristic is particularly amenable to MDA-based code generation, which is efficient at generating repetitive but complicated infrastructure code. However, one capability that current MDA code generation frameworks lack is that they do not provide solutions to generation of a load testing suite.

The aim of our work is to automate the generation of complete benchmark suites from a design description. The input is a UML-based set of design diagrams for the benchmark application, along with a load testing client modeled in the UML 2.0 Testing Profile [4]. The output is a deployable benchmark suite including generated configurations for monitoring/profiling utilities. The target platform is server-side component platforms, currently the Java 2 Enterprise Edition (J2EE) component framework. Executing the generated benchmark application produces performance data in an analysis friendly format, along with automatically generated performance graphs. This approach has a number of benefits:

- The generated benchmark suite is based on a design that closely corresponds to the application of interest, and hence it captures the unique characteristics of the application. This should lead to the benchmark producing more representative measures of the eventual application.
- Model driven code generation hides the complexities of the benchmark implementation from architects, and helps them focus on analyzing the benchmark results that are automatically produced.
- The approach derives the benchmark application from the same application model that the performance analysis model is derived. This makes it possible to use the performance measurement data for analytical model validation and tuning model parameter values. This essentially complements the performance analysis framework proposed in [22].
- Following MDA standards and using existing open source MDA frameworks significantly reduces the learning curve of the approach. It also takes advantage of existing code generation "cartridges" exploiting the latest component technologies. The wide range of interoperable UML modeling tools (due to the MDA/UML compatibility standard) also makes the approach more amenable to adoption in practice.

The paper discusses related work on performance analysis with MDA and benchmark suite generation. We then introduce our approach in section 3. A case study based on EJB component technology is presented in section 4. We briefly discuss the approach in section 5 then conclude our paper in section 6.

2. Related Work

2.1 Performance Analysis with MDA

It has been argued that the MDA approach is a suitable for facilitating performance analysis of large scale enterprise system [23] since it "*permits natural and economical modeling of design and analysis domains and relationships between them, supporting both manual and automatic analysis.*" Some researchers have used MDA in performance analysis [19, 21, 24, 25], and typically focus on deriving analytical models from UML design models. A UML design model with appropriate profiles applied can be transformed into a specific analytical technique based model for further analysis. Most of these model transformations are based on XSLT. The mapping and transformation information is tangled within the XML query and transformation language which has limitations for representing and validating mapping relationships [9].

A more comprehensive framework has been proposed in [22]. It advocates representing analytical performance models in UML using profiles. Thus, deriving analytical models from design models equates to standard model transformation in MDA. This has various standardization, validation and tooling advantages over the using of plain XSLT to transform designs into proprietary models.

Analytical model-based performance analysis can not work without data to populate the model parameters. Populating parameters must utilize techniques such as simulation, estimation based on experience or historical data, and all these require an in-depth understanding of the behavior of the actual system components. These techniques for parameter population encounter difficulties if applications use complex commercial black box components – this is common for most component-based applications running on commercial application servers.

In such applications, measurements in the form of benchmarking [6] and prototyping [14] are used to obtain valuable information for architects. Comprehensive prototyping can however be expensive. Industry standard benchmark results can be used, but the results are specific to the benchmark application itself, making it difficult to sensibly infer performance characteristics of the application under design. Combining model based analysis with small scale benchmarking has been introduced to solve this problem [17]. This approach can work on different levels. Benchmarking and prototyping results can be combined with model-based estimation as a way of building confidence in performance predictions and ruling out potential inaccuracy. Benchmarking and prototyping data can also be directly fed into a performance model for more accurate prediction. Some recent research results have produced positive results [18].

We argue that deriving a benchmark application from the same application design as the performance models are derived has great potential for validating and calibrating performance prediction models. That is one of the motivations of our approach on customized benchmark generation using MDA.

2.2 Benchmark Generation

There are a large number of code generation techniques and generator frameworks [9] that can be used in benchmark suite generation. We choose a model driven approach because code can be generated from designs directly. The semantic links between our generated benchmark suite code and performance analysis models then exists since they derive from the same application design model. Some pioneering work has been done on generating benchmark and prototyping applications using models, as in [13, 14] [16, 20]. However, these have several limitations:

- The code generators for the chosen technologies are built from scratch by the researchers. They do not utilize any extensible generator frameworks, or draw upon the vast pool of exiting code generation “cartridges” for latest technologies that are maintained by an active community. Any change to the chosen target technology or the introduction of a new technology requires significant extra work from the researchers.
- These methods do not follow the MDA standards. Existing industry experience on UML and code generation is therefore not leveraged. This makes the code generation tasks for performance engineering an additional activity for developers to carry out, rather than a step in an incremental development process. Not following MDA may also compromise tool interoperability and semantic model traceability between derived models.
- The load testing part of the benchmark suite can not be comprehensively modeled compared to using the UML 2.0 testing profile. The latter distinguishes different testing elements (e.g. test context, test cases, data pool, data partition) within a testing environment. This makes load test suite modeling more modular, reusable and modifiable.

Our approach directly addresses these limitations, as the remainder of the paper explains.

2.3 Performance Measurement

Performance measurement for a benchmark suite includes end to end response time, throughput measurement and application server related performance measurement. For end to end performance data, the application needs to be instrumented or a profiler used. Both take significant software engineering effort. In our approach, the load testing suite is modeled in the UML 2.0 testing profile. Deployable implementation and performance data collecting utilities are automatically generated. Executing the program causes performance data to be collected as part of the test execution, with performance data displayed in generated graphs.

Server-side component performance related data can typically only be obtained using the particular component container’s management tools. These are currently mostly proprietary. However, this is changing as component containers expose performance related information through programmatically accessible management components. For example, in J2EE, JMX-based (Java Management Extension) resource monitoring is becoming widely used and standardized. In our approach, we exploit this new capability with the J2EE WebLogic application server. We use either configuration files or scripts generated from UML design models using MDA to monitor application server related performance data.

3. Customized Benchmark Generation using MDA

The overall structure of the benchmark generation system and related process workflow is presented in the boxed area in Figure 1. It also shows on the left side of the diagram the relationship with model-based performance prediction.

An application design in UML must be transformed manually or with tool assistance into a customized benchmark UML design model. This design usually represents the core characteristics of the application that are deemed by the architect to be the most performance critical. The criteria for this transformation are beyond the scope of this paper, as they depend on specific needs and characteristics of the project and performance analysis techniques. The benchmark UML design model is then annotated with UML profiles for code generation, and a load testing client is modeled using the UML 2.0 Testing Profile. This UML model is then exported using XMI – this becomes an input to the AndroMDA tool framework. We have extended the AndroMDA framework with a new cartridge to generate a load testing suite and associated performance monitoring functionality. A cartridge is a collection of meta-model definitions (XML files), code generation handlers (code libraries) and templates. It can process model elements that have specified *stereotypes and tagged values* or model elements that meet certain conditions. Along with exploiting existing cartridges, a complete deployable customized benchmark suite can be generated that has the capability for extensive performance measurements. The same application design can also be transformed into an analytical performance model. The measurements obtained by the benchmark suite can be integrated with the performance model and used to validate or improve its results. Semantic mappings and explicit model transformation rules provided by the MDA approach make such integration straightforward and systematic. This step is beyond the scope of this paper.

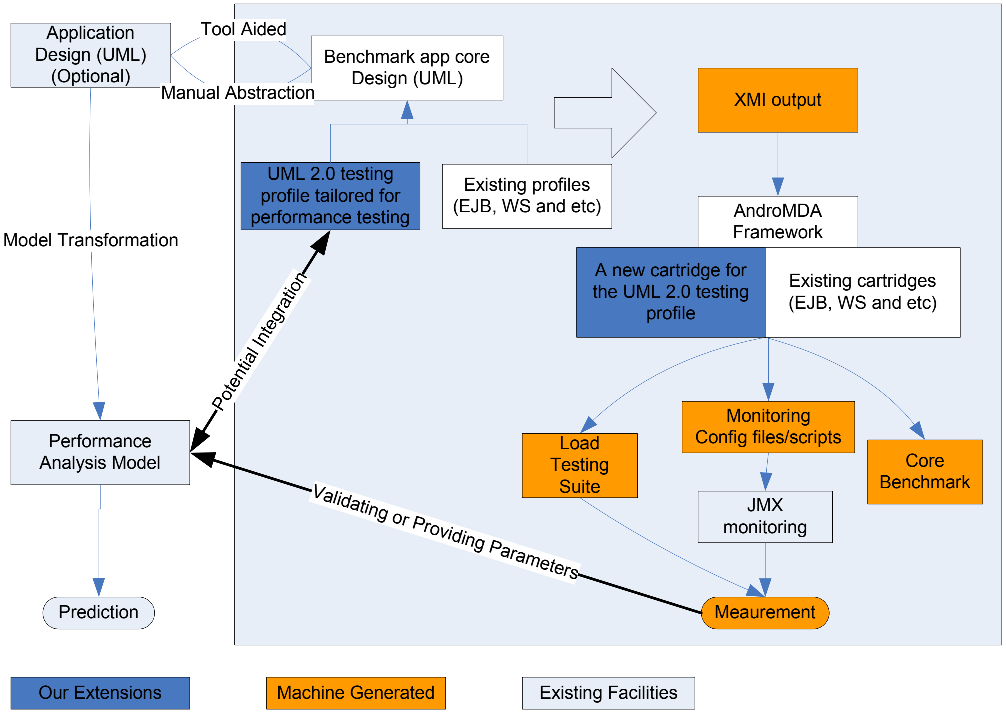
Application Design (UML) (Optional)
Tool Aided
Manual Abstraction
Benchmark app core Design (UML)
XMI output
UML 2.0 testing profile tailored for performance testing
Existing profiles (EJB, WS and etc)
AndroMDA Framework
Model Transformation
A new cartridge for the UML 2.0 testing profile
Existing cartridges (EJB, WS and etc)
Potential Integration
Monitoring Config files/scripts
Load Testing Suite
Core Benchmark
Performance Analysis Model
JMX monitoring
Validating or Providing Parameters
Prediction
Meaurement
Our Extensions
Machine Generated
Existing Facilities

Figure 1. Model Driven Benchmark Suite Generation

3.1 Development Environment

We use an open source extensible framework, AndroMDA [1], for MDA based code generation. The reasons for this are twofold:

- The success of any code generation framework depends on the range of generation capabilities available, along with the extensibility of the framework itself. AndroMDA satisfies both of these requirements. Existing cartridges are maintained by a community of active developers. It also has no intrinsic limitations on extending it to platforms other than the current J2EE and web (service) platforms.
- For a model driven approach to generate deployable components, some implementation details need to be captured at the model level. This makes abstract modeling harder and models cluttered, as models should be a reasonable abstraction devoid of as many details as possible. AndroMDA separates UML and generated code from manually provided business logic in different directories. This allows us to have both code generation capability and abstract modeling at the same time.

We consequently extended AndroMDA to support a subset of the UML 2.0 Testing Profile for load test suite generation. Based on the extended AndroMDA, benchmark designers can model their own benchmark application along with a load testing suite in UML. AndroMDA takes XMI outputs representing the UML design and generates deployable code using available code generation cartridges and our cartridge extensions.

3.2 Benchmark Application Generation

The core benchmark application generation simply exploits MDA development techniques using AndroMDA. The principle behind AndroMDA is to generate as much code as possible from marked PIMs. PIMs are modeled in a UML profile with platform independent stereotypes like “Entity”, “Service” and any necessary persistence information. PIMs can also be annotated with platform specific tagged values.

After modeling, AndroMDA generates all necessary source files including business method interfaces and implementation skeletons. The implementation skeletons are stored in a separate directory so future code re-generation will not override the skeletons.

Modeling and generating the benchmark application is not therefore a distinct engineering step from normal development activities. It can be considered as one of the steps in an incremental development process instead of a throw away performance prototyping activity. One PIM model can also be used to generate different deployable applications for different platforms with little modification. This can greatly reduce the cost and consequently the hurdle of performing performance engineering in practice.

3.2 Load Test Generation

We model the load testing behavior using the UML 2.0 Testing profile [4]. This profile is an OMG standard, representing a comprehensive superset of existing testing widely used frameworks such as JUnit. Currently however, there is no test generator available based on the UML testing profile. The Eclipse Hyades project [5], which will eventually fully support it in terms of modeling (not automatic generation), remains in a nascent state. To develop a full test generation framework according to the profile is a major effort, and beyond our available resources. Hence we focused merely on modeling load tests using a subset of the profile and producing a default implementation of the model including both test logic and test data.

To this end, we have implemented the following stereotypes in the UML 2.0 Testing Profile through extending AndroMDA:

- **SUT** (System under Test): This represents the application to be tested. It consists of one or more objects. The SUT is exercised via its public interface by Test Components. In our approach, the SUT is the entry point for the system (e.g. a remote session bean in J2EE) which will be exercised for load testing purposes only.
- **Test Context**: This is a collection of test cases. In our approach, it consists of load test cases with a default implementation of loadTestAll(), which exercises all the business methods on the SUT.
- **Test Component**: Test components are classes of a test system. A test component has a set of interfaces via which it communicates with the SUT. Since we provide a default implementation of load test cases, users do not have to model their own Test Components. These Test Components are manifested as default method implementations within the loadTestAll() method. Users can choose to model their own Test Components, which will be treated as any normal class communicating with the SUT.
- **Data Pool**: A data pool is a collection of explicit values that are used by a test context or test components during testing. We use a data pool to model the load testing data used when calling each method on SUT. The system will automatically generate random test data based on the data type and range specified. The data pool can also be used to specify the transaction percentage mix for all business methods by using tagged values associated with the stereotype. These configuration capabilities allow users to accurately and easily model the anticipated work load.
- **Data Partition**: Data partitions are logic values for a method parameter used in testing. It typically defines an equivalence class for a set of values. We use a Data Partition to partition the Data Pool into specific sections for load testing. A Data Pool is general enough for other testing purposes, including functional testing.
- **Test Case**: A test case is a specification of one or more test cases for the system. It includes what to test with which inputs, and the results expected under various conditions. Test cases are modeled within a Test Context that consists of multiple test cases. In our approach, a loadTestAll() test case is implemented by default. It exercises all public methods on the SUT using randomly generated data modeled in a Data Pool and a selected Data Partition.

Figure 2 illustrates the subset of the UML 2.0 Testing Profile modeled in MagicDraw 8.0 as metafacade extensions in AndroMDA. They all extend the existing AndroMDA metafacade to take advantage of the underlying framework. Each metafacade also includes a set of tagged values associated with it to specify detailed configurations. (We omit the details of these extensions for clarity). The UML model can be exported as a profile which can be later used in a load testing suite modeled along with the benchmark application business logic.

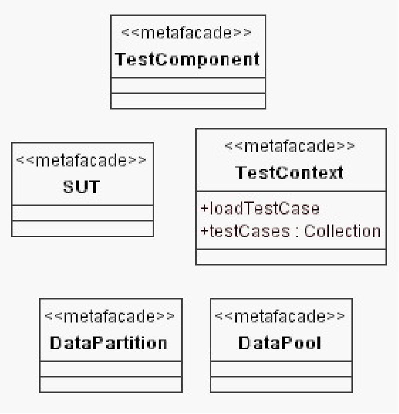

Figure 2. Extending AndroMDA to support UML 2.0 Testing Profile

The AndroMDA extension for load test modeling and generation results in a new cartridge that can be put into the AndroMDA cartridge repository. It works with all other existing server-side generation cartridges. Our cartridge includes some supporting facilities and a library of handlers for code generation written in Java. This facilitates the reuse of existing cartridges and framework facilities. It also enables template writers to access metafacades in templates using tags. The separation of OO based metamodeling with template based code generation provides the flexibility and ease of use.

We provide a complete template for generating a default implementation of the *loadTestAll()* test case with randomly generated data based on a data pool model. A database seeder is also generated to repopulate the database before a new test. These capabilities greatly reduce the extra effort involved in using the suite in load testing activities, in which performance testing is the main interests of the software engineer.

3.3 Performance Data Collection

In the above two sections, we have explained how we have used MDA principles to generate a suite including a core benchmark application and a load testing suite. We also need to collect performance data for either informal analysis or to feed in to a performance analysis model. The data we need falls into two categories, namely:

Application related profiling information:

This includes end to end timings for requests and counts on method calls, CPU workload and memory status, garbage collection counts and so on. Much of this data must be obtained through running a profiler and system utilities. In our default load testing implementation, the end to end response time distribution, average response time and throughput are automatically recorded on the client side. A response time distribution graph for each run is also automatically generated. This data is crucial for performance analysis, and is easy to obtain if the default implementation is used. Users can also use any profiler to obtain similar information.

Application server (middleware) related information

This includes health indicators of the application server itself and component related information that may have performance implications. For example, in EJB, the component pool, runtime cache status and data source pool can be inspected. This information is typically collected through a container-proprietary user interface and with logging and export functions, and the effort required varies immensely between different EJB platform implementations.

As mentioned in section 2.3, several J2EE containers have recently exposed container related information using JMX *MBeans* for both API based and scripting access.

With the help of JMX based container monitoring, we can automate the container performance monitoring using script and configuration file generation. The generation templates only need to be written once for each container and can be reused for different application designs.

In our approach, two styles of accessing the information are available:

- We can use the JMX Java API to query the MBeans directly. A generic MBean information query and collection program runs independently of the benchmark application, and reads a generated configuration file. The configuration file includes the name of the MBean providing the performance information, methods to be called and all necessary parameters. The data collecting program is J2EE container specific and needs to be developed once for each container. With MDA, this requires a generation template. The configuration file can then be automatically generated from the UML model for different application designs. Figure 3 is a snippet of such a generation template.
- A more convenient solution is to leverage the scripting shell provided by some J2EE containers. In this case, no JMX-based data collection program is involved. All that needs to be generated is appropriate scripts for collecting the measurement of interests. For example, third party applications like wlshell [3] for Weblogic JMX and the newly integrated WLST [7] for Weblogic 9.0 provide a scripting shell for accessing the information in MBeans. Figure 4 is an example of the generated scripts for collecting EJBPoolRuntime performance information.

We provide both shell script and configuration file generation templates for the Weblogic platform. We assume by default the target of the performance measurement will be all the beans in the container and business methods supported. Architects can tailor this by commenting out unwanted parts in the generated files.

```
 # methods for Session and MDB Pool
<XDtMethod:forAllMethods>
<XDtMethod:methodName/>:<XDtMethod:methodNameWithout
Prefix/>
</XDtMethod:forAllMethods>
</XDtClass:forAllClass>
…
<XDtEjbSession:forAllBeans>
EjbPoolRuntime:<XDtEjbSession:concreteFullClassName/>
</XDtEjbSession:forAllBeans>
```

Figure 3. A snippet of configuration generation template

```
#Get Runtime info for Session Bean and MDB
ls /EJBPoolRuntime
ejbpool = $LAST
for $ejb in $ejbpool do
   sb1 = get /EJBPoolRuntime/$ejb/AccessTotalCount
   sb2 = get /EJBPoolRuntime/$ejb/IdleBeansCount
   sb3 = get /EJBPoolRuntime/$ejb/BeansInUseCurrentCount
end
```

Figure 4. JMX-based server side monitoring using scripts

4. Case Study

We use the Stock-Online system [10] as a case study to illustrate our approach. The Stock-Online system is a proven benchmark for evaluating various middleware platforms. The original system was developed for different J2EE platforms. Due to platform differences, there was significant effort involved in implementing the same design for different platforms, and keeping the benchmark application in line with component technology advancements required significant ongoing effort. Hence, it makes sense to use Stock-Online as a case study to demonstrate the amount of effort that can be saved utilizing MDA development and code generation.

The case study's aim is to generate the Stock-Online system, including a load testing suite, purely from UML models. To this end, two variations are generated, for the WebLogic and JBoss platforms respectively, both using the same Oracle database backend.

4.1. Benchmark Application Modeling

The server side logic is modeled using the UML and AndroMDA profiles shown in Figure 5. Domain components corresponding to persistent entities are marked using the stereotype <<Entity>>. Components which act as business process facades are marked using the stereotype <<Service>>. There are also tagged values associated with each stereotype for component attributes such as persistence, remote/local interface and other configuration settings. Dependencies among <<Entity>> and <<Service>> elements are marked using <<EntityRef>> if a reference exists between them. All of these values are J2EE platform independent.

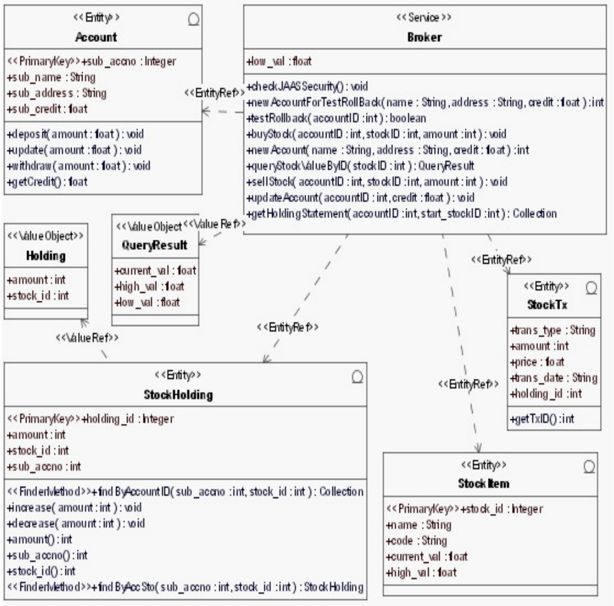
<<Entity>>
Account
<<PrimaryKey>>+sub_accno : Integer
+sub_name : String
+sub_address : String
+sub_credit : float
+deposit(amount : float) : void
+update(amount : float) : void
+withdraw(amount : float) : void
+getCredit() : float
<<EntityRef>>
<<Service>>
Broker
+low_val : float
+checkJAASSecurity() : void
+newAccountForTestRollBack(name : String, address : String, credit : float) : int
+testRollback(accountID : int) : boolean
+buyStock(accountID : int, stockID : int, amount : int) : void
+newAccount(name : String, address : String, credit : float) : int
+queryStockValueByID(stockID : int) : QueryResult
+sellStock(accountID : int, stockID : int, amount : int) : void
+updateAccount(accountID : int, credit : float) : void
+getHoldingStatement(accountID : int, start_stockID : int) : Collection
<<ValueObject>>
Holding
+amount : int
+stock_id : int
<<ValueObject
QueryResult
+current_val : float
+high_val : float
+low_val : float
<<Value Ref>>
<<EntityRef>>
<<Entity>>
StockTx
+trans_type : String
+amount : int
+price : float
+trans_date : String
+holding_id : int
+getTxID() : int
<<ValueRef>>
<<EntityRef>>
<<EntityRef>>
<<Entity>>
StockHolding
<<PrimaryKey>>+holding_id : Integer
+amount : int
+stock_id : int
+sub_accno : int
<<FinderMethod>>+findByAccountID(sub_accno : int, stock_id : int) : Collection
+increase(amount : int) : void
+decrease(amount : int) : void
+amount() : int
+sub_accno() : int
+stock_id() : int
<<FinderMethod>>+findByAccSto(sub_accno : int, stock_id : int) : StockHolding
<<Entity>>
StockItem
<<PrimaryKey>>+stock_id : Integer
+name : String
+code : String
+current_val : float
+high_val : float

Figure 5. Stock-Online benchmark application server side modelling

We omit these tagged values in the presentation to reduce cluttering. In this case study, we do not use any platform specific features, so no platform specific annotations need to be included on the UML model. Hence the same design model is used for the both target platforms.

4.2 Modeling Load Testing Suite

The load testing suite for Stock-Online is modeled using a subset of the UML 2.0 Testing Profile with support from our extension to AndroMDA. The load testing suite model for Stock-Online is shown in Figure 6.

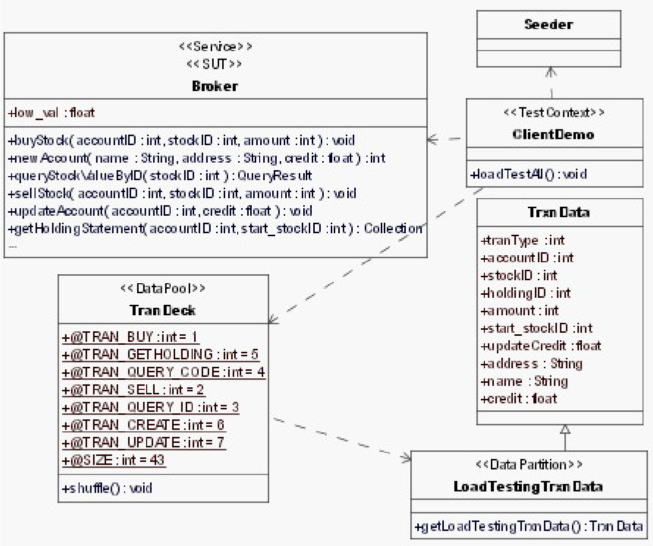
Seeder
<<Service>>
<<SUT>>
Broker
+low_val : float
+buyStock(accountID : int, stockID : int, amount : int) : void
+newAccount(name : String, address : String, credit : float) : int
+queryStockValueByID(stockID : int) : QueryResult
+sellStock(accountID : int, stockID : int, amount : int) : void
+updateAccount(accountID : int, credit : float) : void
+getHoldingStatement(accountID : int, start_stockID : int) : Collection
...
<<TestContext>>
ClientDemo
+loadTestAll() : void
TrxnData
+tranType : int
+accountID : int
+stockID : int
+holdingID : int
+amount : int
+start_stockID : int
+updateCredit : float
+address : String
+name : String
+credit : float
<<DataPool>>
TranDeck
+@TRAN_BUY : int = 1
+@TRAN_GETHOLDING : int = 5
+@TRAN_QUERY_CODE : int = 4
+@TRAN_SELL : int = 2
+@TRAN_QUERY_ID : int = 3
+@TRAN_CREATE : int = 6
+@TRAN_UPDATE : int = 7
+@SIZE : int = 43
+shuffle() : void
<<Data Partition>>
LoadTestingTrxnData
+getLoadTestingTrxnData() : TrxnData

Figure 6. Stock-Online benchmark load test model

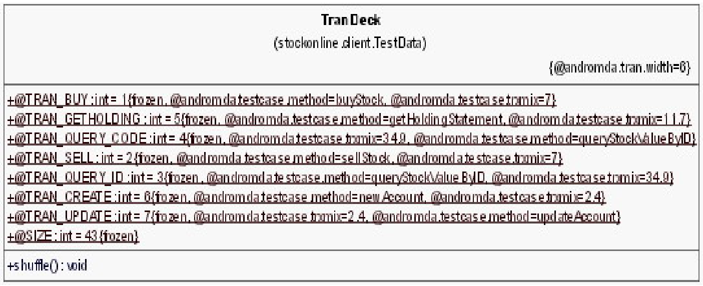
TranDeck
(stockonline.client.TestData)
{@andromda.tran.width=6}
+@TRAN_BUY : int = 1{frozen, @andromda.testcase.method=buyStock, @andromda.testcase.trxmix=7}
+@TRAN_GETHOLDING : int = 5{frozen, @andromda.testcase.method=getHoldingStatement, @andromda.testcase.trxmix=11.7}
+@TRAN_QUERY_CODE : int = 4{frozen, @andromda.testcase.trxmix=34.9, @andromda.testcase.method=queryStockValueByID}
+@TRAN_SELL : int = 2{frozen, @andromda.testcase.method=sellStock, @andromda.testcase.trxmix=7}
+@TRAN_QUERY_ID : int = 3{frozen, @andromda.testcase.method=queryStockValueByID, @andromda.testcase.trxmix=34.9}
+@TRAN_CREATE : int = 6{frozen, @andromda.testcase.method=newAccount, @andromda.testcase.trxmix=2.4}
+@TRAN_UPDATE : int = 7{frozen, @andromda.testcase.trxmix=2.4, @andromda.testcase.method=updateAccount}
+@SIZE : int = 43{frozen}
+shuffle() : void

Figure 7. Transaction type and percentage mix modeled using <<Data Pool>>

The load testing entry point is the *Broker* bean. It is the front end component of the system under test, which is marked using <<SUT>>. *ClientDemo* is the <<TestContext>> which consists of test cases. Only the default *loadTestAll()* test case is included with its default implementation to be generated. For simplification, all the test data is modeled in *TrxnData* from which <<DataPartition>> *LoadTestingTrxnData* is derived. In more complicated situations, several test data classes may exist. In <<DataPool>> *TranDeck*, we can also indicate the transaction mix percentage as tagged values shown in Figure 7. For example, *queryStock* represents 34.9% of all transactions and *getHolding* represents 11.7%. This data will be used in randomly generating test data which simulates the real work load.

We then export both the core benchmark application and load testing suite UML diagrams into XMI compatible formats. Since we are not using any application server specific modeling information in our model, the exported UML model will be used for both WebLogic and JBoss code generation.

4.3 Customized Benchmark Generation, Deployment and Execution

By running the AndroMDA wizard, two EJB project directory structures are generated for WebLogic and JBoss respectively. It consists of a MDA directory for storing the exported UML model, and directories for storing source code and the future deployable application. Project property files for specifying dependencies on the targeted platforms and other deployment configurations are also generated. We then copy the exported UML model into the designated MDA directory and run the code generation engine.

Source code is generated based on the UML model. For the client side, the complete load testing suite is generated without the need for further modification. The load testing logic and random test data is derived from the load testing UML model and method signatures of the server-side component interface.

The server-side component code is generated following EJB best practices, including value objects. Business logic inside each component method needs to be manually added by placing implementation code into a separate directory. This prevents overriding manual modification by subsequent code generation iterations. It also separates the implementation from the specifications (interfaces) which derive from the UML design. After adding the server side business logic code, the AndroMDA framework generates the deployable package. The generated application is almost a carbon copy of the manually written Stock-Online system [10].The application is then deployed in the following environment:

Table 1. Hardware and Software Configuration

Machine	Hardware	Software
Client	Pentium 4 CPU 2.80 GHz, 512M RAM	Windows XP, JDK1.4
Application and database server	Xeon Dual Processors, 2.66 GHz, HyperThreading enabled, 2G RAM	WindowsXP Prof. JDK1.4 with settings – hotspot, –Xms512m and –Xmx1024m, Oracle 9i and thin JDBC driver

4.4 Performance Output

The benchmark test workload, defined by the number of concurrent requests, is specified in the workload generator. We then run load tests and obtain the following performance results automatically:

- Response time distributions in both log files and chart based graphs
- Average response time for each request
- Application throughput in terms of transactions per sec.

Figure 8 shows example outputs of response time distribution chart for two application servers under a workload of 100 concurrent clients.

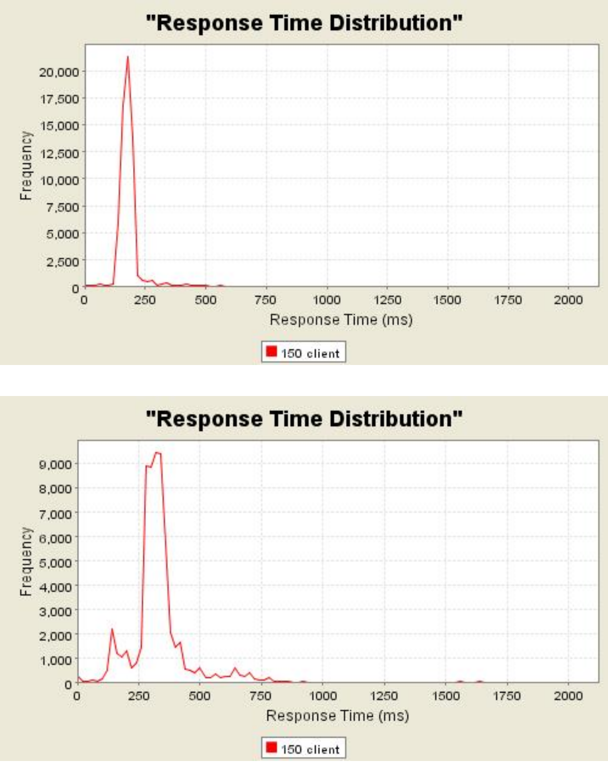

Figure 8. Samples of response time distribution on two application servers

By utilizing the JMX-based monitoring and data collection techniques described in section 3.3, many server side internal parameters can be obtained. For example, Figure 9 shows the performance parameters automatically collected from benchmark execution for 100 clients on WebLogic application server with 20 server threads and 20 database connections.

The performance data collected in this example is the same as we have been collecting in conducting various empirical studies on evaluating architecture candidates and platforms. Interested readers can refer to [17] for the details of the type of data collected and how to interpret the data for performance comparisons. Using the technology described in this paper, we now have a tool for automatically generating the core application and load testing components of benchmarks following MDA, and efficiently capturing the performance data.

- **JDBC Connection Pool**
  - PrepareStatement cache hit ratio = 99.999%
  - Current # of active connections = 19
  - # of pending requests = 0
  - Average # of active connections = 12
  - Maximal # of active connections = 20
- **Application Server Worker Thread Pool**
  - Current # of idle threads = 0
  - Current # of pending requests = 79
  - # of requests completed = 43,000
- **Entity Bean Cache**
  - Cache hit ratio = 69%

Figure 9. Parameters collected based on automatically generated JMX monitoring configurations

5. Discussions

5.1 Evaluation

It took considerable effort to develop the original Stock-Online application for different middleware platforms, conduct load testing and collect performance data. Using our extended framework, one student, the main developer of the AndroMDA extension but with no experience with J2EE, took one week to model the system and conduct the load tests on WebLogic. It took her another half day to deploy the system on JBoss and conduct the same load testing. We realize that this effort comparison it is anecdotal, and we do not have accurate effort data from the original development team. It is also easier to develop the same application the second time. However, it is preliminary evidence that the effort spent on the basic *plumbing* code and load test suite has been absorbed by cartridge developers and code generation. Hence we believe the productivity savings afforded by this approach are potentially considerable. We are planning to conduct a thorough investigation on this.

In our case study, we are not using any platform specific features so one single UML model is used for both the JBoss and WebLogic platforms. If platform specific features are involved, the model has to be marked with platform specific information using tagged values. The additional complexity of using the platform specific features is also hidden by the code generation cartridges.

Making changes to the design to generate a new version of the benchmark is also simpler. If new business components and methods are introduced, new implementation code needs to be added to the generated skeleton. Otherwise, changing design elements like stateful/stateless beans or remote/local interfaces are simply a matter of changing the UML model. All the changes will be propagated in the source code and deployment information.

Generating a benchmark application is essentially not different from generating any software application. A benchmark application usually either captures the performance critical parts of the full application or acts as a way of exercising off-the-shelf component framework services which you are interested in. By unifying benchmark related performance work with normal application development in a single consistent environment, we facilitate incremental design and development. This helps to incorporate performance engineering tasks into the SDLC with minimal cost.

5.2 Limitations

There are still several limitations of this approach:

- AndroMDA is generic enough to support platforms other than the existing J2EE and web (service) platforms. Still, the existing cartridges are limited to these two different type of platforms. The lack of .NET platform support has limited us from applying our approach to a wider collection of platforms. However, new third party vendors such as ArcStyler have successfully provided MDA support for .NET. On the other hand, Microsoft will integrate model based modeling in its new Visual Studio, along with potential interoperability with UML. Another possibility therefore is that a UML model can be transformed into models usable in Visual Studio for code generation.
- The default implementation of the load testing suite is still relatively simple. It covers only successful testing scenario generation. In real applications, performance of exception handling and transaction rollback is also a major concern. Currently, users have to implement such scenarios manually. We are considering integrating these both a modeling level and in the default implementation in future versions. We will also take advantage of some other interesting concepts in UML 2.0 Testing Profile such as Arbiter and Verdict stereotypes for determining the success of the test run.

6. Conclusion and Future Work

This paper has presented an approach to generate customized benchmark application based on the J2EE platform using MDA. An implementation of this approach based on extending the open source MDA framework has been described and demonstrated.

A benchmark design is modeled with platform independent models in UML. A corresponding load testing suite is modeled following a subset of the UML 2.0 Testing Profile. Deployable code is then generated for both the core benchmark design and its associated load testing suite. The core application is generated by taking advantage of existing code generation cartridges maintained by the open source community. The load testing suite generator has been developed by the authors, and fully integrates with the core application generation. A case study using EJB component technology and the Stock-Online benchmark suite is demonstrates the tools and the generated outputs from load tests.

This approach has several significant advantages over proprietary model-based CASE tool environments for benchmark generation. Using MDA and exiting open source MDA frameworks reduces the learning curve and training effort required, and improves model traceability and tool interoperability. A default implementation and test data generation saves a large amount of effort for normal load testing. The approach also complements existing model-based performance prediction methods by providing the potential to use the benchmark results for calibrating and validating analytical performance models. The semantic traceability achieved through MDA makes this integration easier, and this remains a major objective of our future work.

7. Acknowledgments

National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

8. References

[1] "AndroMDA, v3.0M3", http://andromda.org/.

[2] "Model Driven Architecture", http://www.omg.org/mda/.

[3] "wlshell, v.2.0.2", http://www.wlshell.net/.

[4] "UML 2.0 Testing Profile Specification", http://www.omg.org/cgi-bin/doc?ptc/2004-04-02.

[5] "Eclipse Hyades", http://www.eclipse.org/hyades/.

[6] "ECperf, v1.1", http://java.sun.com/j2ee/ecperf/index.jsp.

[7] "WebLogic Scripting Tool (WLST)", http://e-docs.bea.com/wls/docs90/config_scripting/index.html.

[8] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," Software Engineering, IEEE Transactions on, vol. 30 (5), pp. 295-310, 2004.

[9] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in Proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architectures, 2003.

[10] I. Gorton, Enterprise Transaction Processing Systems: Putting the CORBA OTS, Encina++ and OrbixOTM to Work: Addison-Wesley, 2000.

[11] I. Gorton and A. Liu, "Evaluating the performance of EJB components," Internet Computing, IEEE, vol. 7 (3), pp. 18-23, 2003.

[12] I. Gorton, A. Liu, and P. Brebner, "Rigorous evaluation of COTS middleware technology," Computer, vol. 36 (3), pp. 50-55, 2003.

[13] J. Grundy, Y. Cai, and A. Liu, "Generation of distributed system test-beds from high-level software architecture descriptions," in Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE), 2001.

[14] J. Grundy, Z. Wei, R. Nicolescu, and Y. Cai, "An environment for automated performance evaluation of J2EE and ASP.NET thin-client architectures," in Proceedings of the Australian Software Engineering Conference (ASWEC), 2004.

[15] G. T. Heineman and W. T. Councill, Component-based software engineering: putting the pieces together. Boston: Addison-Wesley, 2001.

[16] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen, "Design a Test Suite for Empirically-based Middleware Performance Prediction," in Proceedings of the TOOLS Pacific, 2002.

[17] Y. Liu, "Performance Prediction of Component-based System," in School of Information Technologies. PHD Dissertation: University of Sydney, 2004.

[18] Y. Liu and I. Gorton, "Accuracy of Performance Prediction for EJB applications: A statistical analysis," in Proceedings of the Software Engineering for Middleware (SEM), 2004.

[19] A. D. Marco and P. Inverardi, "Compositional Generation of Software Architecture Performance QN Models," in Proceedings of the 4th Working IEEE / IFIP Conference on Software Architecture (WICSA), 2004.

[20] M. J. Rutherford and A. L. Wolf, "A case for test-code generation in model-driven systems," in Proceedings of the The second international conference on Generative programming and component engineering, Erfurt, Germany, 2003.

[21] M. J. Rutherford and A. L. Wolf, "Integrating a Performance Analysis Kit into Model-Driven Development," in Proceedings of the The 5th GPCE Young Researchers Workshop 2003, Erfurt, Germany, 2003.

[22] J. Skene and W. Emmerich, "Model Driven Performance Analysis of Enterprise Information Systems1," Electronic Notes in Theoretical Computer Science, vol. 82 (6), pp. 1-11, 2003.

[23] J. Skene and W. Emmerich, "A model-driven approach to non-functional analysis of software architectures," in Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE), 2003.

[24] T. Weis, A. Ulbrich, K. Geihs, and C. Becker, "Quality of service in middleware and applications: a model-driven approach," in Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference(EDOC), 2004.

[25] C. Yilmaz, A. M. Memon, A. A. Porter, A. S. Krishna, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Preserving distributed systems critical properties: a model-driven approach," Software, IEEE, vol. 21 (6), pp. 32-40, 2004.