An Approach for Performance Test Artefact Generation for Multiple Technologies from MARTE-Annotated Workflows

Antonio García-Domínguez and Inmaculada Medina-Bulo
Department of Computer Languages and Systems
University of Cádiz
Cádiz, Spain
{antonio.garciadominguez, inmaculada.medina}@uca.es

Mariano Marcos-Bárcena
Department of Industrial Design and Mechanical Engineering
University of Cádiz
Cádiz, Spain
mariano.marcos@uca.es

***Abstract*—Obtaining the expected performance from a workflow would be easier if every task included its own specifications. However, normally only global performance requirements are provided, forcing designers to infer individual requirements by hand. In previous work we presented two algorithms that automatically inferred local performance constraints in Unified Modelling Language activity diagrams annotated with the Modelling and Analysis of Real-Time and Embedded Systems profile. In this work, we present an approach to use these annotations to generate performance test cases for multiple technologies, linking a performance model and an implementation model with a weaving model. We describe how it can be applied to Java code and to Web Service compositions, using existing open source technologies and discussing the challenges involved. The resulting processes follow a meet-in-the-middle approach, allowing the user to write their software according to their needs.**

***Keywords*-software performance; Web Services; MARTE; model weaving; model driven engineering.**

I. Introduction

Software needs to meet both functional and non-functional requirements. Performance requirements are among the most commonly used non-functional requirements, and in some contexts they can be just as important as functional requirements. In addition to soft and hard real-time systems, Service Oriented Architectures (SOAs) must be considered as well. Within SOAs, it is common practice to sign Service Level Agreement (SLAs) with external services, to compensate consumers in case of problems. It is also quite common to create "service compositions", which are services that integrate several lower level services (normally, Web Services from external providers). However, it may be difficult to establish what performance level should be required from the composed services. Too little, and the performance requirements for the composition will not be met. Too much, and the provider may charge more than desired. In addition, developers must test the external services to ensure that they can provide the required performance levels.

There is a large variety of proposals for estimating the required level of performance and measuring the actual performance of a system [1]. Measurements can be used for detecting performance degradations over time, identifying load patterns or checking the SLAs. However, the requirements set by the SLA are usually broad and cover a large amount of functionality: when violated, it might be hard to pinpoint the original cause. Ideally, we should have performance requirements for every part of the system, but that would be too expensive for all but the most trivial systems.

In our previous work [2], we presented two inference algorithms for performance annotations in workflow models. These algorithms can "fill in the blanks" for the response time and throughput requirements of every activity in the model, starting from a global annotation and some optional local annotations set by the user. Users would then write the actual performance tests manually, taking the results produced by these algorithms as a reference. However, writing these tests for every part of a reasonably-sized system could incur in a considerable cost: ideally, it should be partly automated.

In this work, we will outline how to reduce the effort involved in using the results produced by these algorithms by assisting the user in producing concrete performance tests. The models will be used to generate partial test plans and to wrap existing functional test cases as performance tests. To do so, we will weave the existing performance models with design and/or implementation models, relating the performance requirements with the appropriate software artefacts.

The rest of this work is structured as follows: after introducing the models used, we will describe our general approach for generating tests. We will then show two applications, linking the performance requirements to several kinds of software artefacts, and select several candidate technologies. Finally, we will offer several conclusions to this work, and list some future lines of work.

II. Performance models

This section will present the notation used by the performance algorithms described in [2]. We use standard UML activity diagrams, annotated with a small subset of the OMG Modelling and Analysis of Real-Time and Embedded Systems

(MARTE) profile [3]. MARTE provides both a set of predefined performance metrics and some mechanisms to define new ones. In our case, we are using the predefined performance metrics defined in the Generic Quantitative Analysis Modelling (GQAM) subprofile. GQAM is the basic analysis subprofile in MARTE: the Schedulability Analysis Modelling (SAM) and Performance Analysis Model (PAM) subprofiles are based on it. However, SAM and PAM are outside the scope of our approach.

Figure 1 shows a simple example. Inferred annotations are highlighted in bold:

1) The activity is annotated with a ≪GaScenario≫ stereotype, in which `respT` specifies that every request is completed within 1 second, and `throughput` specifies that 1 request per second needs to be handled.
2) In addition, the activity declares a set of context parameters in the `contextParam` field of the ≪GaAnalysisContext≫ stereotype. These variables represent the time per unit of weight that must be allocated to their corresponding activity in addition to the minimum required time. Their values are computed by the time limit inference algorithm.
3) Each action in the activity is annotated with ≪GaStep≫, using in `hostDemand` an expression of the form $m + ws$, where m is the minimum time limit, w is the weight of the action for distributing the remaining time, and s is the context parameter linked to that action.
   The time limit inference algorithm adds a new constraint to `hostDemand`, indicating the exact time limit to be enforced. The throughput inference algorithm extends `throughput` with a constraint that lists how many requests per second should be handled. As these constraints have been automatically inferred, their `source` attribute is set to `calc` (calculated).
4) Outgoing edges from condition nodes also use ≪GaStep≫ but only for the `prob` attribute, which is set by the user to the estimated probability it is traversed.

III. Overall approach

The model shown in the previous section is entirely abstract: at that level of detail, it cannot be executed automatically. It will have to be implemented through other means.

After it has been implemented, it would be useful to take advantage of the original model to generate the performance test cases. However, the model lacks the required design and implementation details to produce executable artefacts. To solve this issue, several approaches could be considered:

1) The abstract model could be extended with additional information, but that would clutter it and make it harder to understand.
2) On the other hand, the implementation models could be annotated with performance requirements, but this would also pollute their original intent.
3) Finally, a separate model that links the abstract and concrete models could be used. This is commonly known as a *weaving model*. Several technologies already exist for implementing these, such as AMW [4] or Epsilon ModeLink [5].

In order to preserve the cohesiveness of the abstract performance model and the design and implementation models, we have chosen the third approach.

After establishing the required links, the next step is generating the tests themselves. To do so, a regular Model-to-Text (M2T) transformation could be used, written in a specialised language such as the Epsilon Generation Language [6]. In case it were necessary to slightly refine or validate the weaving model before, an intermediate Model-to-Model (M2M) transformation could be added. Figure 2 illustrates the models and steps involved in our overall approach.

IV. Applications

We will now show several instances of the overall approach in Figure 2, using different technologies to assist in generating performance test artefacts in different environments.

A. Reusing functional tests as performance tests

Generating executable performance test cases from scratch automatically will usually require many detailed models and complex transformations, which are expensive to produce and maintain. The initial effort required may deter potential adopters. An alternative inexpensive approach is to repurpose existing functional tests as performance tests. This is the aim of libraries such as JUnitPerf [7] or ContiPerf [8]: we will target these libraries in order to simplify the transformations involved and make the generated code more readable.

Listing 1 shows how JUnitPerf is normally used. The original *TFunctional* functional test suite is wrapped into a *TimedTest* (implemented by JUnitPerf) that checks that every test case in *TFunctional* does not take any longer than 1000 milliseconds. The wrapped test case is wrapped once again with a *LoadTest* (also implemented by JUnitPerf) that emulates 10 users running the test at the same time. In combination, the resulting test checks that each of the 10 concurrent executions of the wrapped test finishes within 1 second.

Listing 2 shows a similar fragment for ContiPerf. Instead of using Java objects, ContiPerf uses Java 6 annotations, which would be easier to generate automatically. The *@PerfTest* annotation indicates that the test will be run 100 times using 10 threads, so each thread will perform 10 invocations. *@Required* indicates that each of these invocations should finish within 1000 milliseconds at most. *@SuiteClasses* points to the JUnit 4 test suites to be reused for performance testing, and *@RunWith* tells JUnit 4 to use the ContiPerf test runner.

In both cases, the code itself is straightforward to generate. However, the generated code must integrate correctly with the existing code. If the code was not produced using a model-driven approach, there will not be a design or implementation model to link to. Instead, we will derive a model of the structure of the existing code using a model discovery tool such as Eclipse MoDisco [9]. Eclipse MoDisco can generate models from Java code such as that shown in Figure 3.

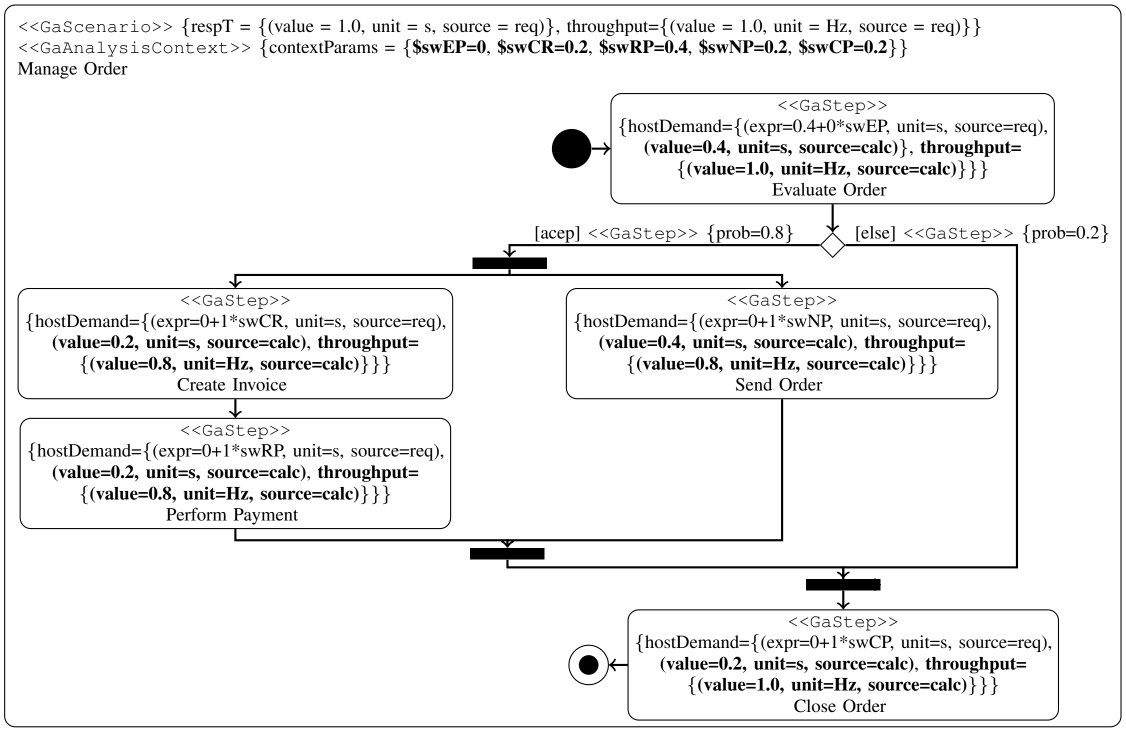

<<GaScenario>> {respT = {(value = 1.0, unit = s, source = req)}, throughput={(value = 1.0, unit = Hz, source = req)}}
<<GaAnalysisContext>> {contextParams = {$swEP=0, $swCR=0.2, $swRP=0.4, $swNP=0.2, $swCP=0.2}}
Manage Order
<<GaStep>> {hostDemand={(expr=0.4+0*swEP, unit=s, source=req), (value=0.4, unit=s, source=calc)}, throughput={(value=1.0, unit=Hz, source=calc)}}}
Evaluate Order
[acep] <<GaStep>> {prob=0.8}
[else] <<GaStep>> {prob=0.2}
<<GaStep>> {hostDemand={(expr=0+1*swCR, unit=s, source=req), (value=0.2, unit=s, source=calc), throughput={(value=0.8, unit=Hz, source=calc)}}}
Create Invoice
<<GaStep>> {hostDemand={(expr=0+1*swNP, unit=s, source=req), (value=0.4, unit=s, source=calc), throughput={(value=0.8, unit=Hz, source=calc)}}}
Send Order
<<GaStep>> {hostDemand={(expr=0+1*swRP, unit=s, source=req), (value=0.2, unit=s, source=calc), throughput={(value=0.8, unit=Hz, source=calc)}}}
Perform Payment
<<GaStep>> {hostDemand={(expr=0+1*swCP, unit=s, source=req), (value=0.2, unit=s, source=calc), throughput={(value=1.0, unit=Hz, source=calc)}}}
Close Order

Fig. 1. Simple example model annotated by the performance inference algorithms

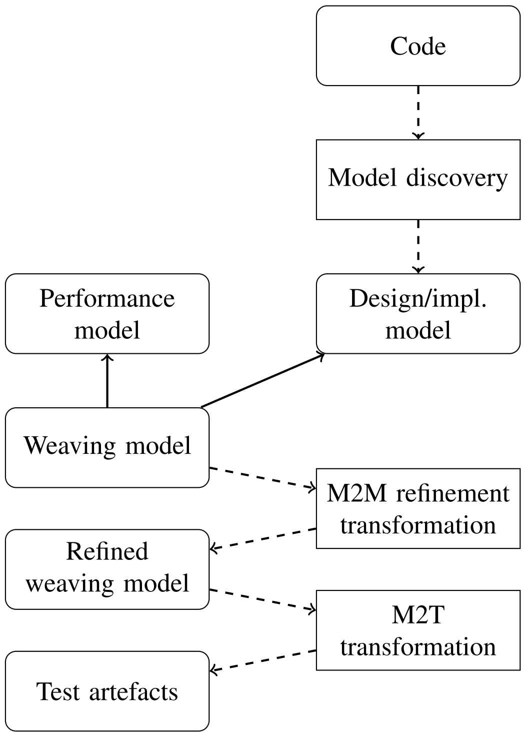


Fig. 2. Overall approach for generating performance test artefacts from abstract performance models

```
final int users = 10;
final int tlimit_ms = 1000;
Test testCase = new TFunctional();
Test test1User = new TimedTest(testCase, tlimit_ms );
Test testAllUsers = new LoadTest(test1User, users );
```

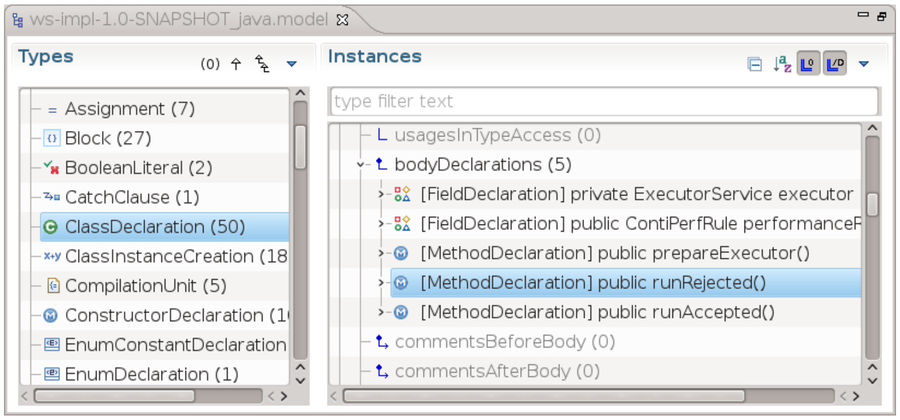
Listing 1. Java code for wrapping the *TFunctional* JUnit 3 test case using JUnitPerf

```
@RunWith(ContiPerfSuiteRunner.class)
@SuiteClasses(TFunctionalJUnit4 . class )
@PerfTest( invocations = 100, threads = 10)
@Required(max=1000)
public class InferredLoadTest {}
```

Listing 2. Java code for decorating the *TFunctionalJUnit4* JUnit 4 test suite using ContiPerf

Once we have the performance and the implementation models, the next step is to link them using a new *weaving model*. Each model consists of an instance of *WeavingModel*, which contains a set of *Link*s between a ≪GaStep≫ stereotype of the MARTE performance model, and a *MethodDeclaration* of the MoDisco model. We can populate the weaving model using the standard Epsilon Modeling Framework (EMF) editors or using Epsilon ModeLink (as in Figure 4).

After linking both models with the weaving model, the last step is running a M2T transformation to produce the actual performance test artefacts. The generated code would be similar to that in Listings 1 or 2.


ws-impl-1.0-SNAPSHOT_java.model

Types (0)

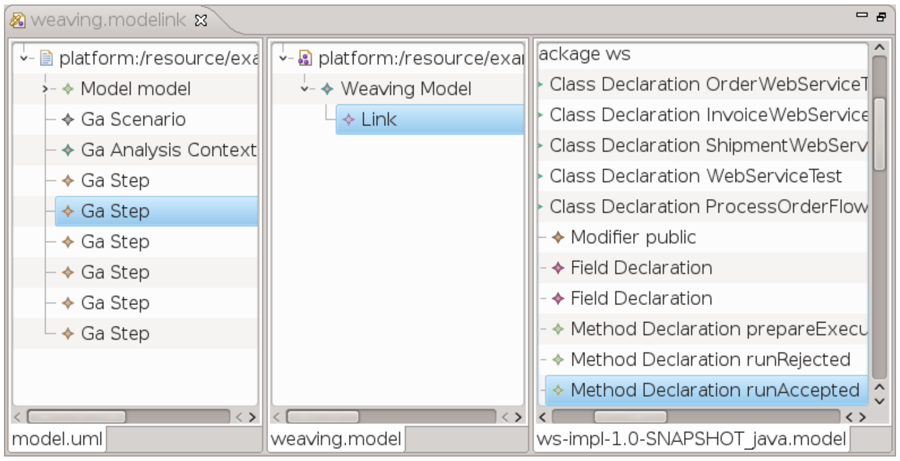
- Assignment (7)
- Block (27)
- BooleanLiteral (2)
- CatchClause (1)
- ClassDeclaration (50)
- ClassInstanceCreation (18
- CompilationUnit (5)
- ConstructorDeclaration (1
- EnumConstantDeclaration
- EnumDeclaration (1)

Instances

type filter text

```
usagesInTypeAccess (0)
bodyDeclarations (5)
  [FieldDeclaration] private ExecutorService executor
  [FieldDeclaration] public ContiPerfRule performanceR
  [MethodDeclaration] public prepareExecutor()
  [MethodDeclaration] public runRejected()
  [MethodDeclaration] public runAccepted()
commentsBeforeBody (0)
commentsAfterBody (0)
```

Fig. 3. MoDisco model browser showing a model generated from an Eclipse Java project


weaving.modelink

```
platform:/resource/exa
  Model model
    Ga Scenario
    Ga Analysis Context
    Ga Step
    Ga Step
    Ga Step
    Ga Step
    Ga Step
    Ga Step
model.uml
```

```
platform:/resource/exa
  Weaving Model
    Link
weaving.model
```

```
ackage ws
  Class Declaration OrderWebServiceT
  Class Declaration InvoiceWebService
  Class Declaration ShipmentWebServ
  Class Declaration WebServiceTest
  Class Declaration ProcessOrderFlow
    Modifier public
    Field Declaration
    Field Declaration
    Method Declaration prepareExecu
    Method Declaration runRejected
    Method Declaration runAccepted
ws-impl-1.0-SNAPSHOT_java.model
```

Fig. 4. Screenshot of the Epsilon ModeLink editor weaving the MARTE performance model and the MoDisco model

```
@WebService
public class HelloWorld {
  @WebMethod
  public String greet(@WebParam(name="name")
  String name)
  {
    return "Hello␣" + name;
  }
}
```

Listing 3. Java code using JAX-WS to implement a “HelloWorld” Web Service

B. Partial test plan generation for Web Services

In the previous section, we applied our approach to existing JUnit test cases, repurposing them as performance test cases. In this section we will discuss how to generate performance test artefacts for a Web Service (WS) [10] in a language agnostic manner.

Web Services based on the WS-* technology stack are usually described using a Web Services Description Language (WSDL) [11] document. This XML-based document is an abstract and language-independent description of the available operations for the service and the messages to be exchanged between the service and its consumers. Existing Web Service frameworks such as Apache CXF [12] can generate most of the code required to implement and consume the services from the WSDL description. Users only need to implement the business logic of the services. In addition, some frameworks (CXF included) can work in reverse, generating WSDL from adequately annotated code. Listing 3 shows an example fragment of Java code that implements a simple “Hello world” Web service using standard JAX-WS [13] annotations.

Since a WSDL document is a declarative and language-independent description of the Web Service itself, we can use it as our design model. After transforming automatically the XML Schema description of the WSDL document format into a regular ECore metamodel [14], we will be able to load WSDL documents as regular Eclipse Modeling Framework models, reusing most of the technologies mentioned in Section IV-A.

The weaving model needs to relate the ≪GaStep≫ stereotypes with the operations of the services in the WSDL document. For instance, we might want to ensure that every invocation of the *evaluate* operation of the *Order* processing

```
grinder.processes=5
grinder.runs=100
grinder.processIncrement=1
grinder.processIncrementInterval=1000
```

Listing 4. Example `.properties` file with configuration parameters for the workload

```
class TestRunner:
  def __call__(self):
    def invoke():
      response = HTTPRequest().POST(
        "http://localhost:8080/orders",
        "(... SOAP message ...)")
      stats = grinder.statistics.getForCurrentTest()
      stats.success = (response.statusCode != 200
                       and stats.time < 150)
      test = Test(1, "Query order by ID").wrap(invoke)
      test()
```

Listing 5. Example Jython script for The Grinder with the contents of the performance test to be run by each simulated client

service finishes within a certain time while handling a certain number of requests per second.

After weaving the WSDL-based model with the performance model, the next step is generating a test plan for a dedicated performance testing tool such as The Grinder [15]. Using a dedicated tool allows for defining tests with less cost and in a way that is independent of the implementation language of the software under test.

In the case of The Grinder, we would need to generate two different files: a `.properties` file indicating several parameters of the workload to be generated, and a Jython script with the test to be run by each simulated client. Listings 4 and 5 show simple examples for these two files. The `.properties` file in Listing 4 indicates that 5 processes should each run the test 100 times, starting with 1 process and adding one more every 1000 milliseconds. On the other hand, the test itself consists of sending an appropriate SOAP message to a specific URL and checking that the response has the OK (200) HTTP status code and that it was received within 150 milliseconds. Since these inputs are quite concise, we deem it feasible to generate an initial version of both files, letting the user add a meaningful SOAP message later.

Later iterations of this application could generate larger parts of the test plan by assisting the user in producing the messages themselves. Links in the weaving model could allow the user to specify a certain strategy for generating the messages to be sent, such as random testing, variations upon a predefined template or static analysis of the code implementing the service. The strategy could be applied in the weaving model refining step showed in Figure 2.

V. Related work

According to Woodside et al. [1], performance engineering comprises all the activities required to meet performance requirements. These activities include defining the requirements, analysing early performability models (such as layered queuing networks [16] or process algebra specifications [17]) or testing the performance of the actual system. Our previous work in [2] focused on helping the user define the requirements using MARTE-annotated [3] UML activity diagrams as notation. The present work is dedicated to helping the user create the performance test artefacts.

Our work does not deal directly with the implemented system, but rather with a simplified representation (a *model*). There is a large number of works dealing with model-based testing, i.e., "the automatable derivation of concrete test cases from abstract formal models, and their execution" [18]. Most of them (as evidenced by [18] itself) are dedicated to functional testing: we will focus on those dedicated to model-based performance testing.

Barna et al. present in [19] a hybrid approach, which uses a 2-layered queuing network (LQN) to derive an initial stress workload for a website. This workload is used to test the system and refine the original LQN model in a feedback loop that searches for the minimum load that would make the system violate one of its performance constraints. Like our work, it combines the analysis of a model with the execution of a set of test cases. However, its goal is completely different: we intend to define the appropriate quality service levels for the individual services in order to meet the desired quality service level of the entire workflow, whereas this approach would estimate the maximum workload that a workflow could handle within a certain quality service level.

Di Penta et al. show in [20] another approach with the same goal of finding workloads that induce service level agreement violations. However, they use genetic algorithms instead of a LQN model and test WSDL-based Web Services instead of a regular website.

Suzuki et al. have developed a model-based approach for generating testbeds for Web Services [21]. SLA and behaviour models are used to generate stubs for the external services used by our own service. This allows users to check that their own services can work correctly and with the expected level of performance as long as the external services meet their SLAs. However, this approach does not generate input messages for the services themselves. Still, we could use this work to check the validity of the performance constraints inferred by our algorithms in [2] in combination with the approach in Section IV-B, by replacing all services in the workflow with stubs and testing the performance of the composition.

As illustrated by the above references, there is a wealth of methods for generating performance test cases and testbeds for Web Services. However, we have been unable to find another usage of model weaving for generating performance test artefacts for multiple technologies. This is in spite of the fact that model composition using model weaving has been used regularly ever since the authors of the original ATLAS Model Weaver proposed it [4]. For instance, Vara et al. use model composition to decorate their extended use case models with additional information required for a later transformation [22].

VI. Conclusion and future work

In this work, we have described an overall approach for generating performance test artefacts from the abstract performance models produced by our inference algorithms in [2]. To generate concrete test artefacts while keeping the abstract performance models separated from any design or implementation details, we propose linking the performance model to a design or implementation model using an intermediate *weaving model*. If a design or implementation model is not available, it can be extracted from the existing code. The weaving model can be then optionally refined using a model-to-model transformation, and finally transformed into the performance test artefacts with a model-to-text transformation.

We have performed an initial study of the feasibility of the approach by studying how to apply it in two situations. The first application will reuse existing JUnit test cases as performance test cases with JUnitPerf and ContiPerf. The implementation model is extracted from the Java code implementing the test cases using the model discovery tool MoDisco [9], and the weaving model links the MARTE annotations in our performance model to the Java test methods in the MoDisco model.

The second application will generate test plans for an independent load testing framework, such as The Grinder [15]. In this case, the WSDL description of the service serves as an explicit design model, and the weaving model links the MARTE performance requirement to an operation of the service. Later revisions of this approach may use the weaving model to specify a strategy for generating the required input messages, instead of leaving it up to the user.

Our next work is to further these feasibility studies by implementing the required transformation workflows. We have implemented a considerable part of the first approach already using MoDisco and Epsilon ModeLink, and we are currently implementing the code generation step in the Epsilon Generation Language.

Acknowledgements

This work was funded by the research scholarship PU-EPIF-FPI-C 2010-065 of the University of Cádiz, the MoDSOA project (TIN2011-27242) under the National Program for Research, Development and Innovation of the Ministry of Science and Innovation (Spain) and by the PR2011-004 project under the Research Promotion Plan of the University of Cádiz.

References

[1] M. Woodside, G. Franks, and D. Petriu, “The future of software performance engineering,” in *Proc. of Future of Software Engineering 2007*, 2007, pp. 171–187.

[2] A. García-Domínguez, I. Medina-Bulo, and M. Marcos-Bárcena, “Model-driven design of performance requirements with UML and MARTE,” in *Proceedings of the 6th International Conference on Software and Data Technologies*, vol. 2. Seville, Spain: SciTePress, Jul. 2011, pp. 54–63.

[3] Object Management Group, “UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) 1.0,” http://www.omg.org/spec/MARTE/1.0/, Nov. 2009, last checked on 2012-03-03.

[4] M. D. Del Fabro, J. Bézivin, and P. Valduriez, “Weaving models with the eclipse AMW plugin,” in *Proceedings of the 2006 Eclipse Modeling Symposium, Eclipse Summit Europe*, Esslingen, Germany, Oct. 2006.

[5] D. S. Kolovos, “Epsilon ModeLink,” 2010, last checked on 2012-03-03. [Online]. Available: http://eclipse.org/epsilon/doc/modelink/

[6] D. S. Kolovos, R. F. Paige, L. M. Rose, and A. García-Domínguez, “The Epsilon Book,” 2011, last checked on 2012-03-03. [Online]. Available: http://www.eclipse.org/epsilon/doc/book

[7] M. Clark, “JUnitPerf,” Oct. 2009, last checked on 2012-03-03. [Online]. Available: http://clarkware.com/software/JUnitPerf.html

[8] V. Bergmann, “ContiPerf 2,” Sep. 2011, last checked on 2012-03-03. [Online]. Available: http://databene.org/contiperf.html

[9] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, “MoDisco: a generic and extensible framework for model driven reverse engineering,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Antwerp, Belgium, Sep. 2010, pp. 173–174.

[10] H. Haas and A. Brown, “Web services glossary,” World Wide Web Consortium, W3C Working Group Note, Feb. 2004, last checked on 2012-03-03. [Online]. Available: http://www.w3.org/TR/ws-gloss/

[11] World Wide Web Consortium, “WSDL 2.0 part 1: Core Language,” http://www.w3.org/TR/wsdl20, Jun. 2007, last checked on 2012-03-03.

[12] Apache Software Foundation, “Apache CXF,” Nov. 2011, last checked on 2012-01-15. [Online]. Available: https://cxf.apache.org/

[13] Java.net, “JAX-WS reference implementation,” Nov. 2011, last checked on 2012-03-03. [Online]. Available: http://jax-ws.java.net/

[14] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed., ser. Eclipse Series. Addison-Wesley Professional, Dec. 2008.

[15] P. Aston and C. Fizgerald, “The Grinder, a Java Load Testing Framework,” 2012, last checked on 2012-03-03. [Online]. Available: http://grinder.sourceforge.net/

[16] D. C. Petriu and H. Shen, “Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications,” in *Proc. of the 12th Int. Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 2002)*, ser. Lecture Notes in Computer Science. London, UK: Springer Berlin, 2002, vol. 2324, pp. 159—177.

[17] M. Tribastone and S. Gilmore, “Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile,” in *Proc. of the 7th Int. Workshop on Software and Performance*. Princeton, NJ, USA: ACM, 2008, pp. 67–78, last checked on 2012-03-03. [Online]. Available: http://portal.acm.org/citation.cfm?id=1383569

[18] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing,” Working Paper 04/2006, Apr. 2006, last checked on 2012-03-03. [Online]. Available: http://researchcommons.waikato.ac.nz/handle/10289/81

[19] C. Barna, M. Litoiu, and H. Ghanbari, “Model-based performance testing (NIER track),” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 872–875.

[20] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, “Search-based testing of service level agreements,” in *Proceedings of Genetic and Evolutionary Computation Conference*, H. Lipson, Ed. London, United Kingdom: ACM, Jul. 2007, pp. 1090–1097.

[21] K. Suzuki, T. Higashino, A. Ulrich, T. Hasegawa, A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini, “Model-based generation of testbeds for web services,” in *Testing of Software and Communicating Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5047, pp. 266–282.

[22] J. M. Vara, M. V. De Castro, M. Didonet Del Fabro, and E. Marcos, “Using weaving models to automate model-driven web engineering proposals,” *International Journal of Computer Applications in Technology*, vol. 39, no. 4, pp. 245–252, 2010.