Model Transformers for Test Generation from System Models

M. Busch[1], R. Chaparadza[1], Z.R. Dai[1], A. Hoffmann[1], L. Lacmene[1], T. Ngwangwen[1], G.C. Ndem[1], H. Ogawa[2], D. Serbanescu[1], I. Schieferdecker[1], J. Zander-Nowicka[1]

[1]Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany; [2]Hitachi Central Research Laboratory Ltd., Japan

Abstract

The early integration of test development into the system development process becomes more and more important. By doing so, design mistakes and implementation faults can be detected in an early stage of the system design and implementation process, i.e. before the newly developed system is shipped to the customer. This allows for reducing the overall development time and costs significantly. This paper reports on results in integrating testing into a model-based development process. Based on a common MOF-based infrastructure, initial test are automatically derived from the system models. Subsequently, an executable test code can be automatically generated from the test models and being applied to the final system for quality assessment.

Keywords

Test modeling, Test generation, UML, U2TP, MDA, MOF

1 Introduction

The Unified Modeling Language (UML) – a widespread visual modeling language that is standardized by the Object Management Group (OMG) – plays an important role in OMG's Model Driven Architecture (MDA) approach. According to the MDA, software development is based on models that are step-wise refined from a platform independent model (PIM) level describing the pure functioning of a system down to platform-specific models (PSM) that are targeted to concrete target technologies. The mapping between models is based on meta-models and done by transformers that read information from a source model repository and store the information in a target model repository.

A first step towards a better integration of test development into the system development process has been taken by [8]. According to this approach, test components are

developed in parallel with the system components as soon as the interfaces and the overall architecture of the system to be developed are defined. However, since the developers still use another language for system modeling than for test models, there is still a gap between the system and test development. Further investigations have thus been done to improve the integration of the test and system development by adopting the same specification language for system and test modeling.

The new edition of UML (UML 2.0) is very promising to be a common ground for system and test models in order to bridge the gap between system and test development. UML2 is a great improvement of the previous version of UML. In particular, its generic extension mechanism supporting different UML profiles for different purposes and domains is an ideal basis for better integration of test and system modeling. Since UML2 still suffers from missing concepts for describing test models efficiently, the OMG defined a UML2 testing profile (U2TP) in order to better support the specification of test models. This UML profile provides particular means for test modeling, i.e. for describing test objectives and test procedures. As a result, the same language (i.e. UML) can now be used in order to specify system and test models. This eases the early integration of test development and system development.

1.1 The UML 2.0 Testing Profile (U2TP)

The Unified Modeling Language (UML) is a graphical language to support the design and development of complex object-oriented systems. While it is flexible in addressing the major object-oriented concepts, test specification and testing issues are beyond the scope of UML version 1.4. In late 2001, the Object Management Group (OMG) issued a Request for Proposals (RFP) to develop a testing profile for version 2.0 of UML (UML 2.0). A UML profile is a domain specific extension of UML provided using a standardized extensibility mechanism.

The UML 2.0 Testing Profile (U2TP) defines testing concepts, including test context, test case, test component, and verdicts that are commonly used during testing. Behavioral elements from UML 2.0 can be used to specify the dynamic nature of test cases. These include interactions, state diagrams, and activity diagrams. Additional concepts for behavior include several types of actions (validation actions, log actions, final actions, etc.), defaults for managing unexpected behavior, and arbiters for determining the final verdict for a test case. The definition and handling of test data is supported by wildcards, data pools, data partitions, data selectors and coding rules. Timers and time zones are also provided to enable specifications of test cases with appropriate timing capabilities.

U2TP also contains a standalone meta-model as a separate compliance point. This allows non-UML tools to provide an implementation that is consistent with the UML

based profile. The Eclipse TPTP project [10] currently has implemented this standalone model as a basis for their test information model.

The UML 2.0 Testing Profile provides a coherent set of extensions to UML 2.0 that support effective test specification and modeling for black-box testing. This is a significant enhancement to UML to support the testing portion of the system development lifecycle. Meanwhile, the UML 2.0 Testing Profile development has come to its finalization and it has become an official standard of the OMG.

1.2 Using MDA Concepts for Test Modeling

According to the MDA approach [9] defined by the OMG, system development starts with the specification of a platform-independent model (PIM) of the system to be developed[1]. This model specifies the correct functioning of the system independently of platform-specific details which define the implementation and execution of the system on a target platform later on. Due to this platform-independence, there is just one PIM needed for several target platforms. The PIM may be refined in several iterative steps. Once the PIM is finalized, one or more platform-specific models (PSMs) can be derived by appropriate transformers. This is shown in Figure 1. For each PSM, the appropriate transformation step adapts the structure and the functionality of the PIM to a specific target platform. Subsequently, system code for a dedicated target platform can be generated from each PSM by appropriate transformers.

Please note that all transformation steps between PIM and PSM, and between PSM and system code can be done automatically or semi-automatically. In the latter case, the derived PSM need to be completed before the next transformation step can be started. Depending on the completeness of the PSM-to-system-code transformation, the generated system code is ready for execution or still needs to be manually completed.

The most benefit from the MDA approach can be taken if a system to be developed shall be realized by more than one target platform. However, even if there is just one target platform for the system under development planned for the time being, the MDA approach enables the flexible and easy on-demand support for further platforms later on.

[1] There is even one more abstract model: the Computational Independent Model (CIM). However, as experts are still discussing about the abstraction level and the concepts of CIM, this is not yet considered by us.

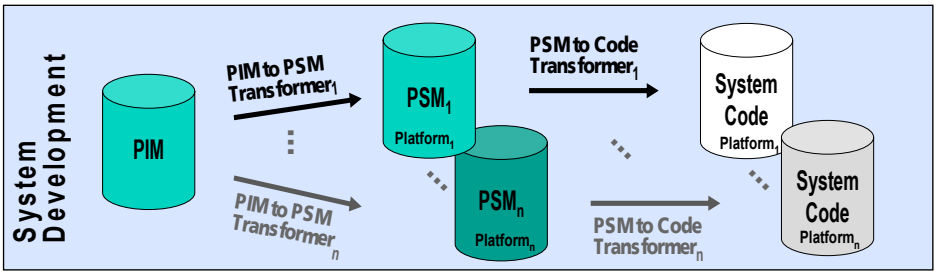

Figure 1: MDA-based system modeling

For test modeling, the same abstraction in terms of platform-independent and platform-specific modeling can be applied as known from system modeling [9]. As shown in Figure 2, from a platform-independent test model (PIT) several platform-specific test models (PSTs) can be derived by proper transformers. From each PST, test code can be generated for the dedicated target platform of the system to be tested and the test execution platform.

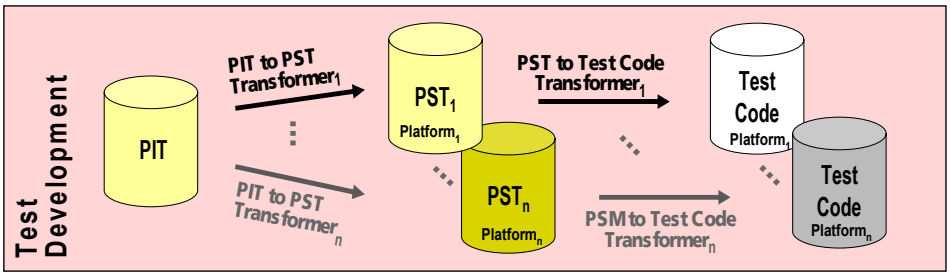

Figure 2: MDA-based test modeling

It should be noted, that it is not always necessary to have a separate PST for each target platform of the system under test. If an abstract test language is used (such as TTCN-3 [3]) and the target platforms are quite similar with respect to its overall nature, the final adaptation to the concrete system's target platform can be done at test code level by proper test adapters.

2 The Approach

The basic idea of deriving test models from system models is to reuse the information about the system to be developed also for developing the test model as the counterpart to the system. In particular, the following system information can be used for the derivation of test models for black-box testing:

- the structure and configuration of the system to be developed in terms of components, interfaces, connected instances, etc.,

- the system behavior externally observable at component ports,
- the type system (in particular user-defined types, e.g. structured types),
- some concrete data values, e.g. used for selecting between branches in the control flow.

2.1 Overview on Test Derivation Transformers

In order to use the information about the system under test for the derivation of test models, dedicated transformers are used. As depicted in Figure 3, those transformers access the system models at platform-independent or platform-specific level and derive appropriate test models/test model skeletons. A platform-specific test model (PIT) can be derived from a PIM; a platform-specific test model (PST) can be derived from a PSM. If the system development provides more than one PSM, several transformers may be applied to the PSMs in order to derive appropriate PSTs. From each PST, test code can be generated for each target execution platform of the system to be tested.

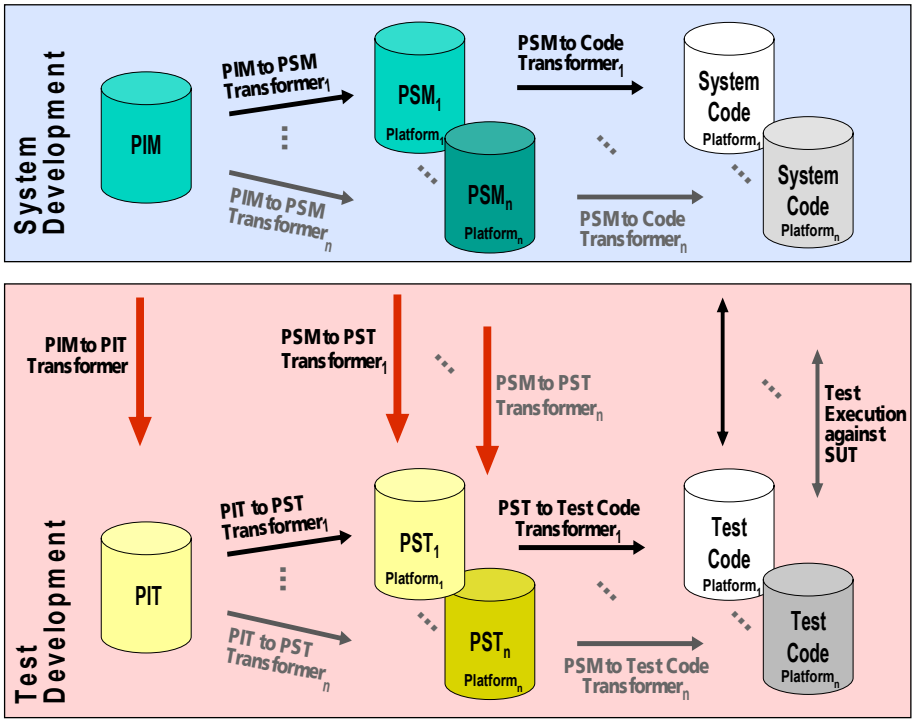

Figure 3: Derivation of test models

2.2 Why is Testing Useful in Case of Automated Code Generation?

This section gives answers to the question, why testing is useful and needed also if the system code is derived automatically from the system model (at PIM or PSM level)? And why it is also useful, if the tests are derived as well from the same model? As to be seen in Figure 3, the test model and the test code as well as the system model and the system code are based on the same originating model information stored in the PIM and/or the PSM(s). However, there are many reasons why testing of the resulting system is still needed.

Incomplete System Model

Complex systems are often not modeled with all their details. Sometimes, complex internal functionality is modeled by single abstract operations since the modeling of complex algorithms would take more effort than implementing them in an appropriate language with dedicated features especially designed for this application area.

Since we focus on black-box testing at system level, it is still possible to derive complete tests from such incomplete system models as the complete internal behavior is not needed for deriving black-box tests. When the tests are executed against the final system code, the internal behavior of the system (not modeled in detail) is automatically implicitly tested (if it is internally needed by the system to produce the output externally observed by the test system).

Invalid Transformers

In most cases the transformers used for deriving platform-specific models and code are not validated. They may thus produce malformed models or code. This will be detected during testing the final generated system code. This is one major reason why testing is still needed even if the source (system) model for system code and test code is the same one.

Incomplete or Parameterized Transformers

Transformers may not be complete, i.e. sometimes they produce model or code skeletons that need to be manually refined and/or completed. Thus, at a later stage of the system development, the system models (at PSM or code level) need to be completed. Whether the refined or completed models or system code behave as expected or not, will be checked by executing the test code against the final system code.

If the transformers at the system development side are parameterised, further information is added during the transformation steps (i.e. selection of options). When testing the system code, it will be checked by the test system, whether those model/transformation changes done by the parameter input/selection break the final system or not.

Changes at Code Level

As mentioned in section 0, particular complex functionality may be needed to be added at code level if the system model does not cover all details required for the system implementation. Also, as pointed out in section 0, if the transformers used do not produce complete system code, the generated code needs to be completed. This modified/completed code must be tested in order to make sure that it behaves like expected.

Even in the case that the model and the code transformers are complete and correct, it may happen that some hand-made changes are done at code level. One reason for this could be to improve the performance. This is another reason why the final system code needs to be tested finally by the test system.

Execution on Real Target Environment

Once the system code is generated from the PSM, it is executed in a concrete target environment. During the system execution, a lot of code is executed that was not modeled (in detail) in the PIM and the appropriate PSMs, such as

- external system or application framework libraries, middleware libraries, etc., and
- operating system functionality.

This external code used by the system code may be malformed and not behave as expected. One reason for this could be incompatibilities between different library versions. As a result, the system code may not behave in the actual target environment as expected and modeled.

Another source of unexpected behavior during execution of the system code in the concrete target environment is the sharing of resources between different parallel clients or components of different instances of the system to be tested or of other applications (being not part of the model). This shared resource usage may lead to serialization of actions originally intended to run in parallel (e.g. serialized due to semaphores). This may break the systems functionality. In the worst case, this interference between different instances or other applications due to joint resource usage may lead to deadlock situations.

2.3 The Tool Chain

Figure *4* provides an overview on the entire tool chain. The upper part of the diagram shows the tool chain used for the system development. Since UML 2 as a whole is too broad and it has a less precisely semantic definition (when it shall be used for automatic transformation or code generation), a subset of UML 2 has been defined together with a specialized clear semantics. This subset is called eUML (essential UML). eUML comes

with a metamodel which formally defines the modelling elements and their relations. The eUML language concentrates on the most important 5 diagram types out of the 13 diagram types supported by UML 2. These diagrams include structure diagrams, activity diagrams and package diagrams as the most important elements.

eUML is extended into the essential Test Modelling Language (eTML) providing concepts needed for test modelling. For eTML, a subset of the most important U2TP concepts has been chosen. Table 1 presents those concepts being grouped into2:

- Test architecture, defining concepts related to test structure and test configuration, i.e. the elements and their relationships involved in a test,
- Test behaviour, defining concepts related to the dynamic aspects of test procedures and addressing observations and activities during a test,
- Test data, defining concepts for test data used in test procedures, i.e. the structures and meaning of values to be processed in a test, and
- Time, defining concepts for a time quantified definition of test procedures, i.e. the time constraints and time observation for test execution.

eTML is used for PIT models, while TTCN-3 (the Testing and Test Control Notation [3]) is used for PST models. TTCN-3 has been chosen because of its abilities to support the automated generation of executable tests for different target technologies. In addition, TTCN-3 tests can be executed locally or remotely also in a distributed manner. This capability of TTCN-3 to execute tests on different platform, for different programming language under different operating systems is in particular valuable for a generic development approach starting with platform-independent models in UML.

Test architecture concepts	**Behavior concepts**	**Data concepts**	**Time concepts**
SUT	Test case	Wildcards	Timer
Test components	Defaults	Data pools	
Test context	Verdicts	Data partitions	
Test configuration		Data selectors	
Test control			

Table 1: The eTML Concepts

eUML models are developed and edited with an eUML add-in for Enterprise Architect [5]. Enterprise Architect (EA) is a graphical UML2 tool that (partially) supports the profiling mechanism of UML2. It is also the graphical front-end tool for eTML.

[2] For further details please refer to [11].

An eUML specification is stored in a Medini [4] repository and is the starting point for deriving test models. The test models are also stored in a Medini repository: an eTML repository for PIT models and a TTCN-3 repository for PST models. One advantage of Medini repositories is that they can be remotely accessed by transformers via CORBA.

With eUML and eTML being dialects of UML2, the system developer and the test developer can use the same base language. Since Enterprise Architect is used for both modeling system models and test models on platform-independent or platform-specific level, the developers can even use the same tools with the same GUI for viewing and editing system and test models.

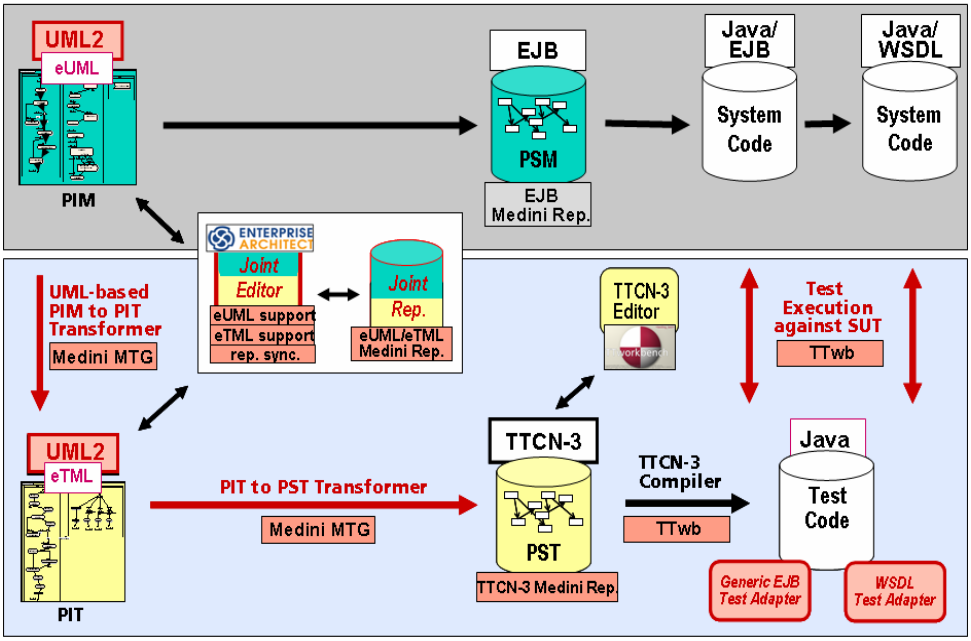

Figure 4: Overview on the tool chain

The Medini Transformer Generator (MTG [4]) is an engine to produce meta-model specific transformer skeletons, which allow for half automated implementation of transformation rules. The generated transformer skeletons provide proper means to read the source repository as starting point for the transformation and to store the new model as a result of the transformation step in the target repository via CORBA. The transformer skeletons contain operations signatures that need to be implemented by the transformer developer in order to implement the transformation rules. It has been used to develop the transformers from PIM to PIT and from PIT to PST. The transformation between

the PST in TTCN-3 and the test code in Java is done by the TTCN-3 compiler of TTworkbench [6].

TTworkbench is a TTCN-3 based integratede development environment (IDE) providing various capabilities for test development and test execution which are integrated in the Eclipse platform. In particular, the Core Language Editor for TTCN-3 is used to visualize and potentially modify the generated TTCN-3 test models. The TTCN-3 compiler is used to generate Java test code that can be executed with the test management and test execution tool TTman being also part of TTworkbench. In result, the tests being derived from system models and potentially further refined on PIT, PST and/or test code level are finally automatically being executed and applied to the system resulting from the system development process.

2.4 Selected Transformation Rules

This section presents selected transformation rules between eUML and eTML. It uses a Pizza Shop example, where pizza can be selected and ordered by customers. For the derivation of eTML diagrams from eUML diagrams, all eUML elements will be adopted to an eTML model first. Since eTML concepts inherit from eUML concepts, all eUML elements are valid eTML elements.

Transformation Rule 1. *All elements of the eUML model become the starting point for the eTML model.*

The following transformation rules are about the generation of the basic test system architecture consisting of package and test contexts.

Transformation Rule 2. *An overall test package is created in parallel to the system model. This test package is named "TestPackagesDiagram".*

Figure 5 shows an example of an overall test package listing the names of its test context classes.

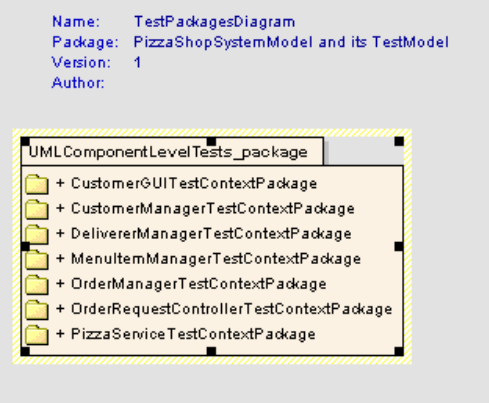

Figure 5: Creating Test Model Package

Transformation Rule 3. *Create a package inside the test package. Name this package according to "UMLComponentLevelTests_package". For each testable UMLClass, a package must be created inside the overall test package/SystemNameSuite.*

Transformation Rule 4. A class stereotyped with <<TestContext>> must be created inside the *UMLComponentLevelTests_package*. The names of the test context classes are "UMLClassNameTestContextPackage".

Figure 6 shows the Package PizzaShopSuites with a Class PizzaService inside, which is stereotyped by <<TestContext>>.

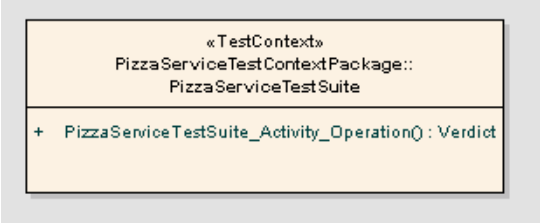

Figure 6. Test Contexts Created for the Pizza-shop Example

Transformation Rule 5. A test configuration for a TestContext is derived from a composite structure diagram if it exists. Herefore, a deployment class must be created, which references to the name of the composite structure diagram (test configuration) for the TestContext.

Transformation Rule 6. The test configuration and the Deployment class associated with the TestContext must be put into the respective package of the TestContext.

Transformation Rule 7. The SUT is derived both from a class or a part. A UML class is considered testable if any of its ports provides an interface, through which operation calls can be accepted by this class, i.e. the class provides services via the interface and one of the interface's operations either returns a value or raises an exception. This also applies to a UML part.

Transformation Rule 8. Test components are derived both from a class or a part. A Test component emulates an entity/component in the system. Therefore, it inherits the system model class from which it is derived in order to get the attributes of the class being emulated.

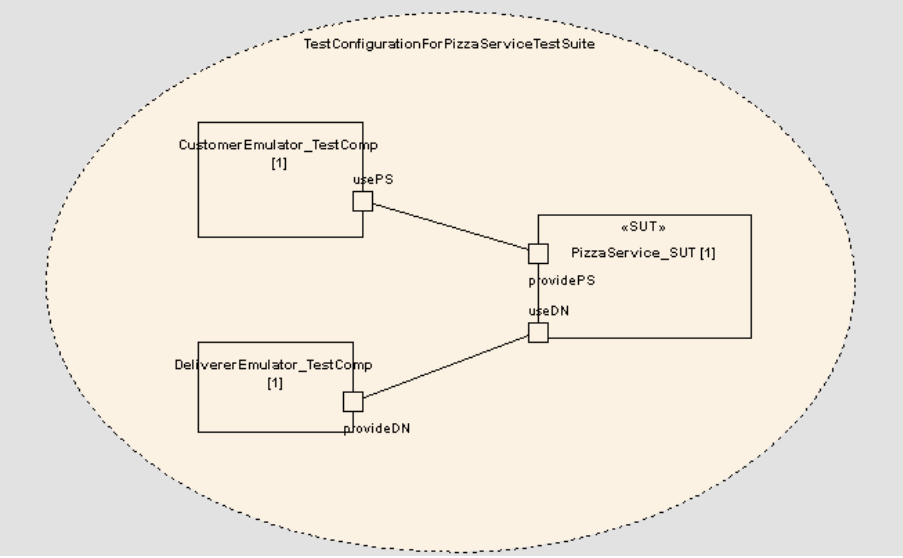

Figure 7: Test Configuration Composite Structure Diagram

In Figure 7, the adopted test configuration is shown. It is a Composite Structure Diagram which has been adopted from the Composite Structure diagram on which the UMLClass exists. In this configuration, the *CustomerEmulator* and the *DelivererEmulator* are test components while the *PizzaService* is assigned to the role of an SUT.

Transformation Rule 9. Traverse the classes of test configuration assigning for each of them <<TestComponent>> or <<SUT>> stereotypes appropriately

so that more Composite Structure Diagrams can be obtained (if needed). The Algorithm of assigning the <<SUT>> stereotype is based on checking if a class is testable.

Transformation Rule 10. Each testable class must create a class stereotyped with <<TestContext>> inside their respective TestContext packages. The number of TestContexts depends on and is equal to the number of testable classes found.

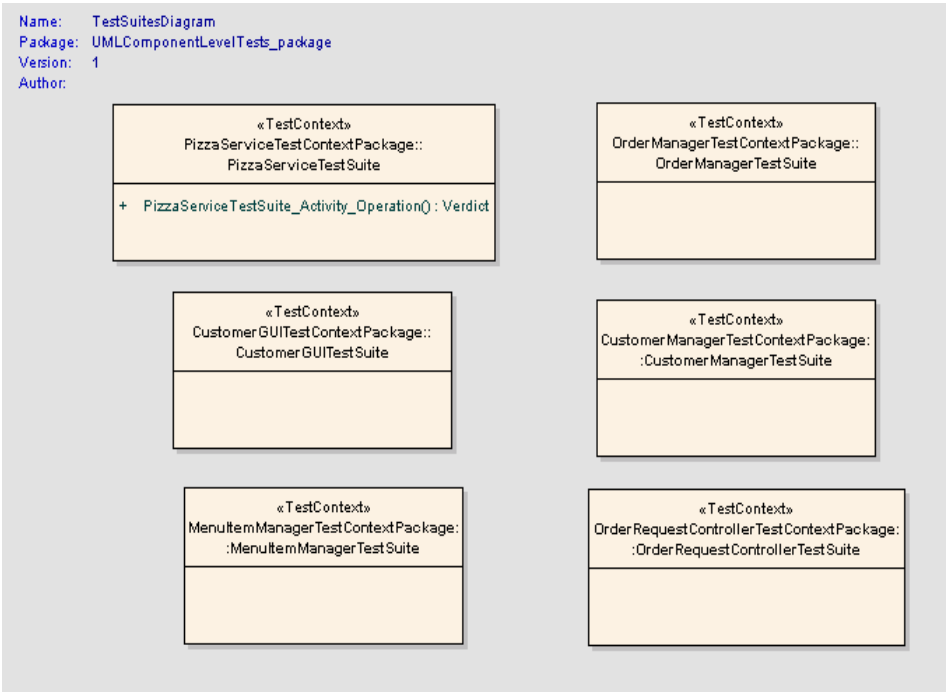

Figure 8: Test Package with different Test Context Classes

Figure 8 presents the test Package with different test contexts inside depending on the potential test configuration. The class on the left side corresponds to the test configuration with SUT being the *PizzaService*, while the class on the right side corresponds to the test configuration with SUT being the *Deliverer*.

A test context defines various test artifacts: attributes, operations, test components, test configuration and test control. Test cases are listed as operations within the test context with verdict as return value. A test context is related to its incorporated test configuration and all test cases which are listed in a test context use the same test configuration. Therefore, for each new test configuration, a new test context is needed.

Every system component that is testable has a test context generated for it. For that, the system components are assigned according to their roles, i.e. SUT or test component.

That means that each system component can be either SUT or test component. For each rotation of the role assignment, the test configuration must be changed. Thus a new test context package must also be created.

Transformation Rule 11. Additional classes have to be created (for each class/part stereotyped with <<TestComponent>> in the Composite Structure Diagram). Those classes inherit from the appropriate classes from the system model. Name these classes according to the rule "ClassNameEmulator_TestComp" where ClassName is the name of the class coming from system model. In this case, ClassNameEmulator inherits from the ClassName.

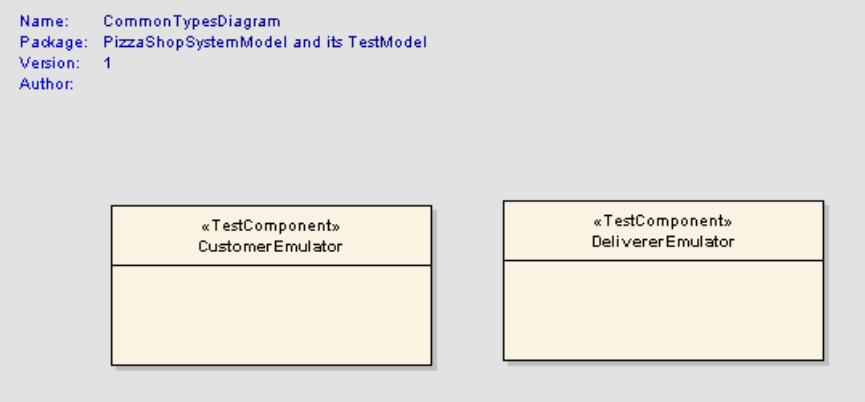

Figure 9: TestComponents in the Test Package

Figure 9 shows two test component classes with ports and their interfaces.

Further transformation rules are needed to complete the mapping from eUML to eTML. Another set of transformations have been defined from eTML to TTCN-3. However, due to lack of space only a selection of transformation rules can be presented here.

3 Results and Summary

This paper discusses model-based testing in the context of MDA and appropriate test languages used for test modeling on platform-independent and platform-specific level. A tool chain to support the activities along model-based testing is presented.

Beginning from PIM and PSM on the system side, PIT and PST are used on the test side. Different approaches to the derivation of test model from the system model are given. The usefulness of test generation from system models where also the system code is derived from is discussed.

Two modeling languages are used for test models: the essential Test Modelling Language eTML (being a subset of the UML 2.0 Testing Profile) and the Testing and Test Control Notation TTCN-3. The execution of the resulting tests is being done via the TTCN-3 platform TTworkbench together with test adapters for EJB and WSDL. This enables us to execute tests automatically for J2EE and WS based systems.

The developed concepts have been applied with the tool chain to e are applied in practice by examples such as the Pizza Shop example. This has been modeled in eUML and tests both in eTML and TTCN-3 are derived. The tests are finally applied to a pizza shop implementation and revealed some errors.

Further work will consist of extending the transformation rules to other additional UML diagrams and to add user control to the model transformers so as to give the user more control to the test generation process.

References

[1] OMG ptc/04-05-02: UML 2.0 Superstructure Specification.
[2] OMG ptc/2004-04-02: UML 2.0 Testing Profile, Final Adopted Specification.
[3] ETSI ES 201 873, v3.1.1: The Testing and Test Control Notation TTCN-3, Standard series, June 2005.
[4] IKV++ Technologies' Medini: www.ikv.de.
[5] Sparx Systems Enterprise Architect: http://www.sparxsystems.com.
[6] Testing Technologies' TTworkbench: www.testingtech.de.
[7] The Hitachi/FOKUS ModTest Project: Model Based Testing Final Report, March 06.
[8] M. Born, I. Schieferdecker, O. Kath and C. Hirai: Combining System Development and System Test in a Model-centric Approach, RISE 2004 International Workshop on Rapid Integration of Software Engineering techniques, November 26, 2004, Luxembourg, Springer LNCS.
[9] OMG: Model Driven Architecture, http://www.omg.org/mda/.
[10] Eclipse: The Test & Performance Tools Platform (TPTP): http://www.eclipse.org/tptp/
[11] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. Williams: The UML 2.0 Testing Profile, Conquest 2004, ASQF Press, September 2004, Nuremberg, Germany.
[12] G. Caplat, J.L. Sourouille: Considerations about Model Mapping, Workshop in Software Model Engineering Oct. 2003, San Francisco, USA, http://www.metamodel.com/wisme-2003/18.pdf.

Presenter's biographies

Due to the number of authors we limit the biographies to the presenter only.

Andreas Hoffmann

Andreas Hoffmann has received his Master's degree in Computer Science from the Humboldt University of Berlin in 1997. Before graduating, he was a member of a project team that developing design tools for distributed applications at FOKUS. Since 1997, he is scientist at FOKUS and has worked in the area of distributed telecommunication systems and modeling and testing thereof. Currently, he is leading the test solu-

tions group at FOKUS. He is also a member of OMG ADTF and IMS Benchmarking SIG.