Automated Generation of Test Cases Using Model-Driven Architecture

A. Z. Javed, P. A. Strooper and G. N. Watson
School of ITEE, The University of Queensland, Australia
{abuzafer, pstroop, gwat}@itee.uq.edu.au

Abstract

In this paper, we demonstrate a method that uses the model transformation technology of MDA to generate unit test cases from a platform-independent model of the system. The method we propose is based on sequence diagrams. First we model the sequence diagram and then this model is automatically transformed into a general unit test case model (an xUnit model which is independent of a particular unit testing framework), using model-to-model transformations. Then model-to-text transformations are applied on the xUnit model to generate platform-specific (JUnit, SUnit etc.) test cases that are concrete and executable.

We have implemented the transformations in a prototype tool based on the Tefkat transformation tool and MOFScript. The paper gives details of the tool and the transformations that we have developed. We have applied the method to a small example (ATM Simulation).

1. Introduction

This paper presents an application of Model-Driven Architecture (MDA) in the context of software verification and validation (V&V).

MDA is an initiative by the OMG (the Object Management Group) to support the development of interoperable, portable and reusable software systems [1]. In MDA, models at various levels of abstraction are the central software design artifact. They are used to facilitate both abstraction and automated development. A simple use of MDA is to model an application in a platform-independent modeling language (e.g. UML). The platform-independent model (PIM) can then be translated into a platform-specific model (PSM) by writing transformation specifications that are mappings between the PIM and some implementation language (e.g. Java) [2]. MDA tools can partially automate the development process by generating most of the code from models resulting in less code to hand-craft [3].

Software V&V is an important quality assurance activity of the software development process. It emerged in the late 1960s as the use of software in military and nuclear-power systems increased [4]. It can play its part throughout the software development life cycle, from requirement specification to actual delivery of the product. An important aspect of V&V is to test the behaviour of a system.

Traditionally software products were verified and validated based upon their specifications [5] or their implemented source code. More recently, model-based testing has become popular [6]. Researchers are investigating the use of software models to support V&V activities. A major advantage of model-based V&V is that it can be easily automated, saving time and resources. Other advantages are shifting the testing activities to an earlier part of the software development process and generating test cases that are independent of any particular implementation of the design.

Not all model-based testing uses MDA. "Model-driven testing" is a form of model-based testing that uses model-transformation technology using models, their meta-models and a set of transformation rules (that are defined in terms of mappings between the elements of meta-models) [7]. The tools based on the model-driven approach automate V&V activities by specifying and executing transformation rules, which reduces the development time and makes their maintenance easier.

We propose a model-driven approach to test software applications using sequence diagrams. Sequence diagrams are behavioural elements of a UML design [8] that describe dynamic interactions among the components of a system. They play an important role in the software development processes that are use-case driven, such as in the Rational Unified Process [9]. Since these descriptions of behaviour are constructed at an early stage, testing based on them can start V&V activities early in the life cycle. An overview of our model-driven approach

Second International Workshop on Automation of Software Test (AST'07)
0-7695-2971-2/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

is shown in Figure 1. The generation of test cases is performed in two steps. In the first step, a UML sequence diagram is translated into a testing model using a horizontal transformation (the transformation that maintains the abstraction level, i.e., PIM to PIM transformation). In the second step, the testing model is converted into a concrete and executable test case using a vertical transformation (PIM to PSM transformation).

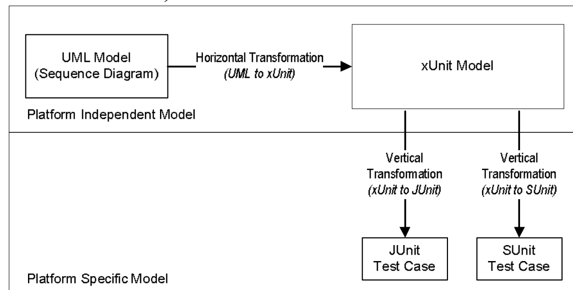

Figure 1: Overall Process

2. Background

The process we propose for the test code generation involves the following tools and technologies: Eclipse Modeling Framework [10], Tefkat [11], MOFScript [12], xUnit [13], JUnit [14] and SUnit [15]. We discuss each of these briefly below.

2.1. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is an MDA tool that facilitates developing model-based applications in Java. It provides an integrated environment for model development, model transformation, and Java code generation, for applications that are based on a structured model. In our method, we use EMF-based transformation engines for generating test cases from a model of UML sequence diagrams.

2.2. Tefkat

Tefkat is an EMF-based model transformation engine which is available as an Eclipse plug-in. It is an implementation of a declarative language which uses meta-models to execute transformation rules that convert a source model into a target model. These rules are mappings between the source meta-model and the target meta-model. The Tefkat transformation engine requires as input the source model, the source meta-model, the target meta-model and a set of rules. It triggers the rules one-by-one on instance(s) of the source meta-model (i.e. source models) and generates instance(s) of the target meta-model (i.e. target models).

Tefkat rules are named and consist of the following parts: FORALL, MAKE, SET and WHERE. The FORALL keyword selects all the objects of a particular type in the source model for which this rule has to be triggered. The MAKE keyword creates an object in the target model for each element of the source model selected. The SET keyword assigns the values to these objects in the target model. The WHERE clause is used in conjunction with the FORALL keyword to filter or refine the selection made by the FORALL keyword. The concrete syntax of the Tefkat transformation language is specified on the Tefkat website [11].

2.3. MOFScript

MOFScript is a model-to-text transformation language generating textual outputs from models based on meta-models, and is available as an Eclipse plug-in. The MOFScript transformation tool requires a source model, a source meta-model and a set of rules that are mappings between the source meta-model and the text to be generated. It executes the rules one-by-one on the input model and generates tailored text from that model.

MOFScript rules add and manipulate text in order to produce the desired textual output. These rules can declare variables, have logical expressions and iterators, and can invoke Java methods. The specification of the MOFScript transformation language is provided on its website [12].

2.4. xUnit, JUnit and SUnit

xUnit is a family of unit-testing frameworks used to write and run repeatable tests for software applications. Developers use these frameworks for developing and executing unit test cases, and for regression testing. Amongst the most popular family members of xUnit are JUnit and SUnit which are unit testing frameworks used for testing Java and Smalltalk applications respectively.

In this paper, we describe a method to generate unit test cases for xUnit family members in general. We then apply our method to generate concrete test cases for JUnit and SUnit in particular.

3. The Methodology

We propose a method to test an application, using a sequence diagram. We do this at two levels. The first is to generate test cases from a sequence of

0-7695-2971-2/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

method calls that are part of the sequence diagram and selected by the tester. Typically, the selected method calls originate from a particular lifeline in the sequence diagram and they appear as method invocations in the generated test case. Note that method invocations that originate from subsequent lifelines are invoked indirectly by the selected method calls. To test that this happens as specified in the sequence diagram, we capture traces during the execution of test cases.

Test results are checked by comparing expected and actual return values of the selected method calls, and by comparing the execution traces with the calls in the sequence diagram. We have two versions of its implementation (for JUnit and SUnit) to show one of the advantages of using a model-driven approach. The JUnit implementation is discussed in this paper.

3.1. Generating Test Cases using MDA

The model-driven approach that we use for generating unit test cases consists of two steps. In the first step, we model a sequence diagram as a sequence of methods calls (SMC) which is then automatically transformed into an xUnit model by applying model-to-model transformations using Tefkat. In the second step, JUnit test cases are generated from the xUnit model by applying model-to-text transformations using MOFScript. This process is shown in Figure 2.

Artifacts 1 and 2 in Figure 2 are the meta-models for a sequence of method calls and xUnit respectively, discussed in Sections 3.1.3 and 3.1.4. The artifacts 3 and 4 are transformation rules, discussed in Sections 3.1.1 and 3.1.2. Artifacts 5 and 6 are the two MDA-tools that we use in our methodology to execute horizontal and vertical transformations (Figure 1). Artifact 7 is the source model of the application from which we generate test cases. Artifact 8 is the xUnit model which is an intermediate output. Tefkat executes horizontal (model-to-model) transformations on the source model (7) to generate this xUnit model (8). Artifact 9 is the test data for the SMC model which specifies parameter values and expected return values of method calls in the sequence diagram. By changing the contents of the test data file, different test cases can be generated for the same sequence diagram. Artifact 10 is a simple text file containing code which is copied at the top of the test case. It can be used to define packages, import classes, etc. which is needed for compiling and executing the generated test cases. Artifact 11 is the final output, produced by MOFScript, which is a concrete and executable unit test case. MOFScript reads the test data (9) and code header (10) while executing the vertical (model-to-text) transformations (4) on the xUnit model (8) to generate the unit test case (11).

These artifacts are generic at different levels. Artifacts 1, 2 and 3 are independent of platform, application and sequence diagrams. Artifact 4 is specific to a platform but independent of application and sequence diagram. All these artifacts do not need to be modified when testing different applications on the same platform.

To test an application from a sequence diagram, the tester must provide the SMC model (7), test data (9) and the code header (10) file. Note that by altering the test data, the same testing scenario can be executed with different test data.

Finally, the intermediate output (8) and the final output (11) are produced by the MDA tool.

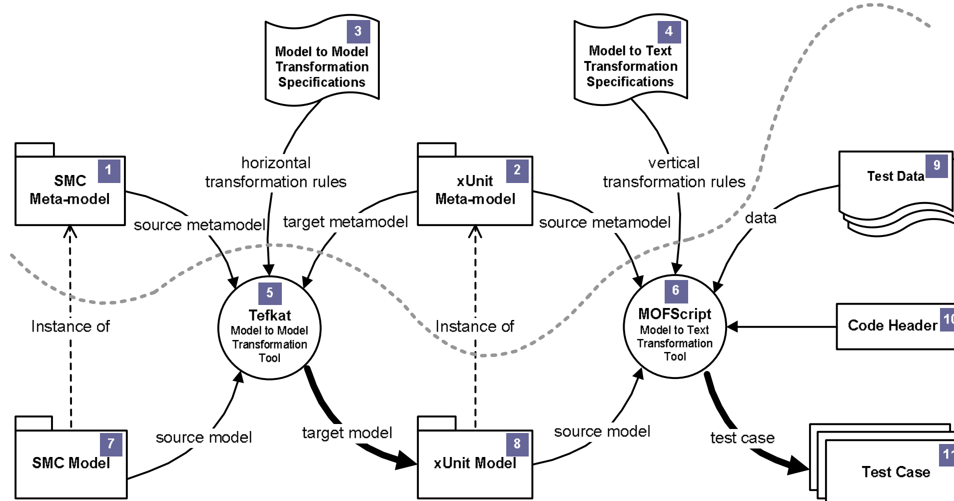

Figure 2: Overview of Methodology

IEEE COMPUTER SOCIETY

3.1.1. Transforming SMC into xUnit. We transform the SMC model into an xUnit model by using Tefkat transformation rules. As an example, the rule in Figure 3 creates a test case in the xUnit model for every SMC in the SMC model. The test case is given the same name as the SMC. All the Tefkat rules that are used in the transformation are available online [16].

```
RULE    SMC_2_TestCase ( smc, testCase )
        FORALL  SMC smc
        MAKE    TestCase testCase
        SET     testCase.name = smc.name ;
```

Figure 3: An Example Tefkat Rule

3.1.2. Generating JUnit from xUnit. MOFScript transformation rules are used to generate JUnit test cases from the xUnit model. Two example MOFScript rules are presented in Figure 4.

```
model.TestSuite::main( ) {
   printf("public class Test_"+self.name+"extends TestCase{\n")
   self.testCase->forEach ( tc:model.TestCase ) {
      tc.mapTestCase( )
   }
   printf( "} // End of Test Suite" )
}
model.TestCase::mapTestCase( ) {
   printf(" \n\r\t public void Test_" + self.name + "( ) { " )
   ...
   printf(" \n\r\t } //End of Test Case" )
}
```

Figure 4: Example MOFScript Rules

The rule model.TestSuite::main is the entry-point rule where the transformation starts. The expression self.name is the name of the object on which the rule is being executed, i.e. the name of the test suite in this case. The forEach keyword iterates over the collection of test cases in the test suite and invokes the rule model.TestCase::mapTestCase to process them. This rule creates a JUnit test case and invokes other rules (that are not discussed in detail) to complete the body of the test case. Together 3.1.1 and 3.1.2 produce a test case for each SMC. So far we have defined MOFScript transformation rules for generating JUnit and SUnit test cases, from the xUnit model. All these rules are available online [16].

3.1.3. SMC Meta-model. As we are using UML sequence diagrams only for generating test cases, we confine our implementation to a meta-model of sequences of method calls, ignoring more complex aspects of sequence diagrams [17] such as connectors, message-occurrence-specifications, message-ends and message-events.

Our simplified meta-model for sequences of method calls is shown in Figure 5. It consists of interactions, messages, classes, parameters, expected values and literal strings. In this model, an Interaction represents part of a sequence diagram. The Messages contained in the interaction are a subset of the method calls of the sequence diagram selected by the tester. The messages can have Parameters and an optional ExpectedValue in them. The parameter and the expected value are of type ScalarValue or ComplexValue. The Scalar Values are atomic data values that do not contain any other data values. Instances of ScalarValues in Java are integer, float, String, etc. The ComplexValues are the values that contain other values i.e. they act as data structures. The examples of ComplexValues in Java are all classes except Strings. Moreover every message is associated with an OwnerClass (that owns the methods being called), which is a class that receives the message. The owner class has parameters for its constructors that are required to create an instance of the class in the generated test case.

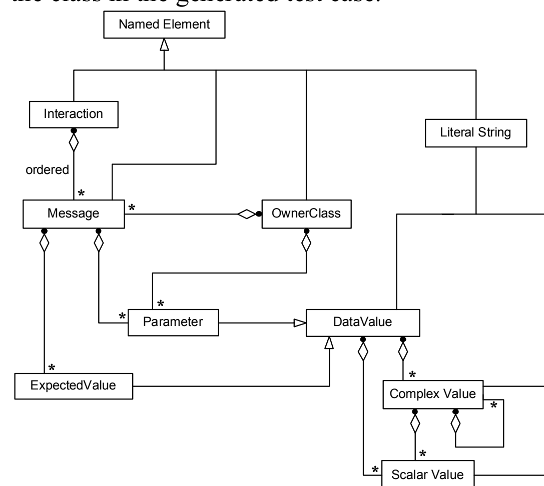

Figure 5: SMC Meta-model

3.1.4. xUnit Meta-model. The meta-model for xUnit test cases is shown in Figure 6. No meta-model for xUnit was available, so we derived it by studying the architecture of test cases written in different unit testing frameworks such as JUnit and SUnit. In this model, the Test Suite acts as a container for Test Case(s). A test case can have Assertions in it. An assertion is a condition that should hold true after executing the test case. An assertion can be of different types which are specified by its attribute type, e.g. the JUnit framework has Equal, Not Equal, Same, Not Same, True and False assertions. For testing of sequence diagrams, an assertion has an expected value and a method call, i.e. the code to be tested.

0-7695-2971-2/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

After executing a test case, the unit testing framework compares the actual value (the value returned after executing the code) with the expected value to decide on the success or failure of the test case. As an example, the JUnit's Equal assertion compares the actual value and the expected value. If both values are equal, the assertion holds. Conversely, the Not Equal assertion holds if the values are not equal. Moreover, the method can have parameters that are either scalar values or complex values as discussed in the meta-model of sequence diagrams. The elements Message, OwnerClass, ComplexValue, ScalarValue and DataValue are the same as in the SMC meta-model.

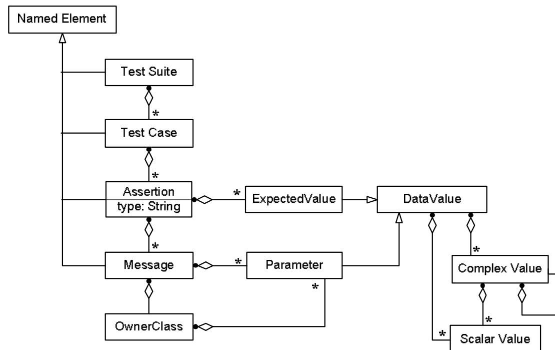

Figure 6: xUnit Meta-model

3.2. Tracing

During the execution of test cases, we monitor the method invocation chain by means of the Daikon [18] tracing tool. We compare the observed method execution chain with the expected method execution chain in the sequence diagram. Currently, we compare the actual traces of the program with the expected traces manually, but this activity can be automated in the future.

4. An Example – ATM Simulation

The tool described in section 3 has been implemented under Eclipse 3.1 on a PC. It has been validated on an ATM simulation [19]. The ATM application simulates an automated teller machine (ATM). An ATM allows its users to perform basic banking operations like withdrawal, deposit, transfer and checking the balance, without having to go to the bank.

We have generated test cases using the tool from the following sequence diagrams: withdrawal, deposit, transfer and balance inquiry. We have executed the generated test cases to test the system. The captured traces were then compared with method invocations in the sequence diagrams. Due to space constraints, we cannot present the details of the example here, but all inputs and outputs (intermediate and final) for the example are available online [16].

5. Discussion

We have described a method for automatic test case generation from UML sequence diagrams. The sequence diagrams are useful for testing because they can initiate software-testing activities in an early stage of the software development process. The method uses model transformation technology that has its own advantages of portability, interoperability, quick development, maintainability, etc. Thus it is an advance over other model-based testing approaches from sequence diagrams, which are based on more traditional technology.

The genericity of our method is extended by targeting xUnit testing frameworks and incorporating an intermediate phase which generates test cases in a platform-independent xUnit format. Thus, potentially, the method can be used to generate test cases for any of the xUnit family members by varying the backend (Artifact 4 in Figure 2). This distinguishes our tool from other tools that generate test cases for a particular platform [20, 21].

Most of the vertical transformation rules (xUnit-SUnit and xUnit-JUnit) have similar logical structure that makes them reusable. They differ in the text which is embedded as language syntax, e.g. for SmallTalk the statement terminator is dot (.) whereas in Java it is semi-colon (;). For the rules having similar structure, the implementer only needs to copy and change the syntax-related text in these rules. Figure 7 shows the reusability in terms of non-commented lines of code that are the same. This reusability is obtained at minimal cost due to the intermediate xUnit model. The structural mapping between SMC and xUnit is addressed during the horizontal transformations leaving the vertical transformations linear and almost identical except for language-related syntax.

The implementation of the tool is cost-effective as the SMC meta-model, the xUnit meta-model and the horizontal transformation are created only once and do not change for different platforms, systems and sequence diagrams (as discussed in Section 3). The implementer needs to provide vertical transformations for the new platform that this tool needs to support. The user (the tester) needs to provide the SMC model, the test data file and the code header to generate test cases using this tool. To generate different test cases for the same SMC, only the test data file needs to be changed.

IEEE COMPUTER SOCIETY

Rule	Structure	Lines of Code			
		SUnit	JUnit	Same	Reuse-%
main	Different	27	25	17	63
mapTestCase	Different	13	20	11	55
mapAssertion	Same	48	47	44	92
mapMessage	Same	32	28	25	78
mapOwnerClass	Same	8	8	8	100
mapParameter	Same	58	56	51	88
mapConstructorParameter	Same	54	52	48	89
mapExpectedValue	Same	36	34	30	83
mapComplexAttribute	Same	28	28	26	93
mapSimpleAttribute	Same	26	25	22	85
Total		**330**	**323**	**282**	**85**

Figure 7: Code Reusability Matrix

We have implemented the method using Eclipse, Tefkat, MOFScript, JUnit and SUnit. This demonstrates the versatility and utility of the MDA approach to software development and tool construction.

We test applications by checking return values of method calls and the method invocation trace. While executing the generated test case, the return values of the methods that are invoked from our test case are checked using assertions. The method calls that these methods make are monitored by examining the trace that is captured during the execution of the test case.

6. Related Work

Model-based testing is gaining support in the software industry [22]. We discuss the related work for generating test case using sequence diagrams in general and using a model-driven approach in particular.

Abdurazik and Offutt [23] propose test criteria based on collaboration diagrams for static and dynamic testing of software systems. Briand and Labiche [24] propose the TOTEM (Testing ObjectorienTed systEms with the unified Modelling language) methodology to derive system test requirements. They derive test requirements by analyzing UML artifacts such as class, use case, and sequence diagrams and OCL constraints across these artifacts but do not generate test cases from these requirements. Wittevrongel and Maurer [20] develop a model-based tool, SCENTOR, which creates functional test drivers for e-business applications from sequence diagrams that have test data (parameters and expected values of method calls) embedded in them. Fraikin and Leonhardt [21] develop another model-based tool, SeDiTeC, which generates test stubs using sequence diagrams that are augmented with test data. These stubs enable testing even before the completion of the system implementation. However, the tools and techniques discussed above do not take advantage of MDA technology.

Pilskalns et al. [25] present an approach to generate test cases from sequence diagrams. They convert a sequence diagram into object method directed acyclic graph (OMDAG) such that its objects become the nodes and its method calls become the edges of the graph. The paths in the OMDAG are augmented with test information (different attribute values and parameter values of methods) that is used to generate cases. Dinh-Trong et al. [26] present another approach and implement a prototype tool, named EPTUD (Eclipse Plug-in for Testing UML Designs) that generates and executes test cases using sequence diagrams. It transforms a UML design model (DUT, design under test) into an executable form (EDUT, executable design under test), adds test scaffolding (TDUT, testable design under test), executes tests and reports failures. However, both these approaches generate test cases that validate the software model, whereas the test cases we generate verify the system implementation.

The OMG itself has standardised UML testing by issuing the UML Testing Profile Specification (U2TP) [27] that defines a language for designing, visualizing, specifying, analysing, constructing and documenting the artifacts of test systems. The U2TP specification also defines a mapping from the UML Testing Profile to JUnit. These mappings are just explanations of U2TP elements in the context of JUnit. For example, U2TP has a Test Case and in JUnit the Test Case is realised as an operation defined in a class inheriting from the JUnit TestCase class. These mapping are not explicit transformation rules from U2TP to JUnit. The work we present provides the actual transformation rules and uses xUnit, which makes the transformation independent of any particular platform.

The work closest to ours is by Dai [28], who discusses the transformation of a UML model into a U2TP model that is a platform-independent test model (PIT). She proposes to generate test cases using three layers of transformations (one horizontal and two vertical) that are UML-PIT (horizontal), PIT-PSM (vertical) and PSM-Test_Case (vertical). However, no tool support is provided for the proposed method.

7. Conclusion

We have proposed a method that generates test cases from the platform-independent model of an application using MDA tools. We devised two sets of

0-7695-2971-2/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

transformations: horizontal transformations using Tefkat and vertical transformations using MOFScript.

We have implemented a prototype tool for generating test cases from sequences of method calls to realise the method. During execution of the test cases, the return values of methods are checked and the method invocation chain is monitored using a tracing tool. Currently, we have two versions of its implementation that are for JUnit and SUnit, but the proposed method is general and can be used to generate test cases for any other xUnit testing framework. The method was applied to an example (ATM Simulation).

In future, the method can be integrated with a UML modelling tool like Rational Rose to read the sequence diagrams to generate test cases. The comparing of the actual trace with the sequence diagram can also be automated. We shall extend our approach to generate integration tests for component-based software using interactions among components of a system as recorded in sequence diagrams.

Acknowledgments

This research is funded by an ARC Discovery grant, DP0557972: Enhancing MDA with support for verification and validation. We would also like to thank Mr. Keith Duddy for various discussion sessions.

References

[1] OMG, "MDA Guide Version 1.0.1," 2004.

[2] J. Poole, "Model-Driven Architecture: Vision, Standards And Emerging Technologies," in Workshop on Metamodeling and Adaptive Object Models, ECOOP 2001.

[3] R. Soley and OMG, "Model driven architecture (2000)," ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf, accessed on 28/2/2007.

[4] R. Dolores and U. Roger, "Software Verification and Validation: An Overview," *IEEE Softw.*, vol. 6, 1989, pp. 10-17.

[5] R. Poston, *Automating Specification Based Software Testing*. IEEE Computer Society Press, 1996.

[6] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Elsevier Inc., 2007.

[7] R. Heckel and M. Lohmann, "Towards Model-Driven Testing," in *Electronic Notes in Theoretical Computer Science*, vol. 82, 2003.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[9] P. Kruchten, *The Rational Unified Process – An Introduction*. Addison-Wesley, 2000.

[10] EMF: http://www.eclipse.org/emf/, accessed on 28/2/2007.

[11] Tefkat: http://tefkat.sourceforge.net/, accessed on 28/2/2007.

[12] MOFScript: http://www.eclipse.org/gmt/mofscript/, accessed on 28/2/2007.

[13] P. Hamill, *Unit Test Frameworks*. O'Reilly Media, 2004.

[14] J. Rainsberger, *JUnit Recepies : practical methods for programmer testing*. Manning Publications Co., 2005.

[15] SUnit: http://sunit.sourceforge.net/, accessed on 28/2/2007.

[16] A. Javed: *http://www.itee.uq.edu.au/~abuzafer/ast07*, accessed on 28/2/2007.

[17] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *Computer*, vol. 39, 2006, pp. 59-66.

[18] Daikon: http://pag.csail.mit.edu/daikon/, accessed on 28/2/2007.

[19] ATM System: http://www.math-cs.gordon.edu/courses/cs211/ATMExample/, accessed on 28/2/2007.

[20] J. Wittevrongel and F. Maurer, "SCENTOR: scenario-based testing of e-business applications," in Proc. of the 10th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001, pp. 41-46.

[21] F. Fraikin and T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," in Proc. of the 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 261-266.

[22] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz, "Model-based testing in practice," in Proc. of the 21st International Conference on Software Engineering, 1999, pp. 285-294.

[23] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation," in Proc. of the 3rd International Conference on UML, 2000, pp. 383-395.

[24] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Softw Syst Model*, vol. 1, 2002, pp. 10–42.

[25] O. Pilskalns, A. Andrews, R. France, and S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations,"in Proc. of the 6th International Conference on the UML, 2003, pp. 234-248.

[26] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews, "A tool-supported approach to testing UML design models," in Proc. of the 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005, pp. 519-528.

[27] OMG ptc/04-04-02: "UML 2.0 Testing Profile."

[28] Z. Dai, "Model-Driven Testing with UML 2.0," in Proc. of the 2nd European Workshop on Model Driven Architecture, 2004.

0-7695-2971-2/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY