Thomas Bauer
Hajo Eichler
Marc-Florian Wendland
Sebastian Wieczorek (Eds.)

Model-based Testing in Practice

3rd Workshop on Model-based Testing in Practice
(MoTiP 2010)

In Conjunction with the

6th European Conference on Modelling Foundations and Applications (ECMFA 2010)

Paris, France, June 16, 2010

Preface

This volume contains the proceedings of the 3rd Workshop on Model-based Testing in Practice (MoTiP) held on 16 June 2010 in Paris, France, in conjunction with the 6th European Conference on Modelling Foundations and Applications (ECMFA 2010).

The objective of the MoTiP 2010 workshop is to bring together industry and academia by providing a platform for interaction and collaboration. The continuing industry trend to raise software complexity by increasing the functionality and accessibility of software and electronic components leads to an ever-growing demand for techniques to ensure software quality. At the same time, software companies are shortening development cycles to respond to the customers demand for fast and flexible solutions. In order to remain competitive, early and continuous consideration and assurance of system quality becomes an asset of ever-increasing importance in industrial software development.

Model-based approaches are not only able to provide effective quality assurance, but also help to evaluate and control the coverage, costs, and risks related to testing efforts. Both – the effectiveness and the efficiency of testing – can be handled by model-based approaches within integrated system and test development for software-intensive systems. While the software industry starts to adopt model-based testing techniques on a large scale, promising research ideas are emerging that have the potential to answer many of today's industrial challenges. Therefore the MoTiP 2010 workshop brings together practitioners and researchers to initiate a dialog much-needed.

The papers and tool descriptions in this volume are representative of current industrial and research activities on Model-based Testing. All selected papers are of high quality, thanks to the professionalism of the authors, reviewers, and program committee members.

We would like to take this opportunity to thank the people who have contributed to the MoTiP 2010 workshop. We want to thank all authors and reviewers for their valuable contributions, and we wish them a successful continuation of their work in this area. Finally, we thank the organization of the ECMFA 2010 conference in which this workshop has been embedded. Special thanks go to Axel Rennoch who was one of the co-founders of the MoTiP workshop series.

June 2010

Thomas Bauer
Hajo Eichler
Marc-Florian Wendland
Sebastian Wieczorek

Organisation

Workshop Chairs

Thomas Bauer	Fraunhofer IESE, Germany
Hajo Eichler	IVU, Germany
Marc-Florian Wendland	Fraunhofer FOKUS, Germany
Sebastian Wieczorek	SAP Research, Germany

Programme Committee

Fevzi Belli	University of Paderborn, Germany
Juhan Ernits	University of Birmingham, UK
Robert Eschbach	Fraunhofer IESE, Germany
Andreas Hoffmann	Fraunhofer FOKUS, Germany
Jacques Kamga	Daimler AG, Germany
Andrei Kirshin	IBM Haifa, Israel
Raluca Lefticaru	Universitatea din Pitesti, Romania
Bruno Legeard	Smartesting, France
Jesse Poore	University of Tennessee, USA
Alexander Pretschner	TU Kaiserslautern, Germany
Christopher Robinson-Mallett	Berner & Mattner, Germany
Ina Schieferdecker	TU Berlin, Germany
Alin Stefanescu	SAP Research, Germany
Dragos Truscan	Abo Akademi University, Finland
Colin Willcock	Nokia Siemens Networks, Germany

Table of Contents

Session 1: Research papers and experience reports 7

Introducing Model-Based Testing in Industrial Context - An Experience Report 9
Hartmut Lackner, Jaroslav Svacina, Stephan Weißleder, Mirko Aigner, and Marina Kresse

Including Model-Based Statistical Testing in the MATERA Approach 19
Andreas Bäcklund, Fredrik Abbors, and Dragos Truscan

Test Case Generation for Product Lines based on Colored State Charts 29
Manal Farrag, Wolfgang Fengler, Detlef Streitferdt, Olga Fengler

Model Based Statistical Testing and Concurrent Streams of Use 39
Frank Böhr

A Trial on Model Based Test Case Extraction and Test Data Generation 43
Xiaojing Zhang and Takashi Hoshino

Session 2: Academic model-based test tools 59

Fokus!MBT – A flexible and extensible toolset for Model-based testing approaches 61
Marc-Florian Wendland

Test Automation as a Model-Driven Engineering Process with MDTester – Test Automation is Model-Driven Software Development 63
Alain-G. Vouffo Feudjio

SIMOTEST: A Tool for Deploying Model-Based Testing in Matlab/Simulink® using IEEE 1641 67
Tanvir Hussain, Robert Eschbach, and Martin Größl

Session 3: Commercial model-based test tools 71

Smartesting TestDesigner 73
Bruno Legeard

Conformiq QTronic 75
Peter Magnusson

SESSION 1
Research papers and experience reports

Introducing Model-Based Testing in Industrial Context – An Experience Report

Hartmut Lackner[1], Jaroslav Svacina[1], Stephan Weißleder[1], Mirko Aigner[2], and Marina Kresse[2]

[1] Fraunhofer Institute FIRST, Department Embedded Systems,
Kekuléstraße 7, 12489 Berlin, Germany
{hartmut.lackner,jaroslav.svacina,stephan.weissleder}@first.fraunhofer.de
http://www.first.fraunhofer.de
[2] Thales Rail Signalling Solutions GmbH,
Colditzstraße 34-36, 12099 Berlin, Germany
{mirko.aigner,marina.kresse}@thalesgroup.com
http://www.thalesgroup.com

Abstract. Model-based testing is an important quality measurement technique. There are several theoretical advantages of model-based testing and experience reports to support them. Model-based testing, however, is not applicable "out-of-the-box". Each environment requires specific adaptations. Thus, there are serious acceptance thresholds in industry. In this paper, we present a report on our efforts to introduce model-based testing as a testing technique in an industrial context.

1 Introduction

Testing is one of the most important system validation techniques. In model-based testing (MBT), the system under test (SUT) is compared to a system specification in the form of a model. Several languages are used to create system models. We focus on UML state machines. A common approach to model-based testing is to generate a test suite based on the system model, to execute the test suite, and to compare the observed behavior of the SUT to the expected one.

Although model-based testing has a high potential for reducing test costs and increasing test quality, this technique is adopted slowly in industrial practice. In our opinion, the major reason for this is that model-based testing is not applicable "out-of-the-box", but requires training and adaptation. This results in costs, e.g. for learning modeling languages, for using model-based test generators, and for integrating model-based testing into the existing testing process. In this paper, we as Fraunhofer FIRST engineers report on our efforts to introduce model-based testing as a new testing technique to Thales Rail Signalling Solutions GmbH during a pilot project.

The paper is structured as follows. In the following section, we present the initial project situation. In Section 3, we present the used toolchains. We describe the course of the cooperation in Section 4 (adaptation and formalization of the system model) and Section 5 (implementation of the test adapter). We evaluate

the used test generation approaches in Section 6. In Section 7, we summarize our results and experiences. Finally, we present related work in Section 8 and conclude in Section 9.

2 Initial Project Situation

In this section, we describe the initial situation of our pilot project. In the project, we focused on testing components of the European Train Control System (ETCS). ETCS is a stationary signaling and train protection system, which is developed as part of the European Rail Traffic Management System (ERTMS). The functionality of the ETCS software components are safety-critical need to be certified (see EN 50128 [1]). Thus, significant effort is applied on quality measurement methods like verification, validation, and test.

According to the regulations resulting from the EN 50128 norm for safety-critical systems, the development process of Thales consists of systematic requirements engineering, functional and design specification, implementation, static analysis methods, and different levels of software testing.

The engineers at Thales use different types of models to specify critical parts of the system: The structure of the system is modeled using class diagrams and the behavior is described using state machines. The models are not used for automatic code generation and several parts of the models are described in an informal way, e.g. using pseudocode and prose. The intention of creating these models was to provide an intuitive semi-formal description of the system behavior and to allow for a common understanding of critical system parts.

At the start of the project, we decided to apply MBT for conformance testing of the system models and the implemented components. The system models were already present but they were not used for code generation. Thus, we chose to reuse them as test models instead of creating new test models from scratch.

3 MBT Toolchains

For automatically generating test suites with MBT, we used one industrial and one academic test tool. In this section, we present the two corresponding toolchains that integrate these tools in the test generation process. Figure 1 depicts both toolchains in an activity diagram: The main element is the system model as the input for both toolchains – the left part shows the industrial toolchain, and the right part shows the academic toolchain.

Both toolchains use the same system model as input and generate test code that is compatible to the test adapter provided by Thales engineers. The following two subsections describe both toolchains in more detail.

3.1 Commercial Toolchain

The commercial toolchain uses two tools and a text transformation program based on Prolog: We used Borland Together [2] for formalizing and concretizing the existing system model. Afterwards, we imported the formalized model to

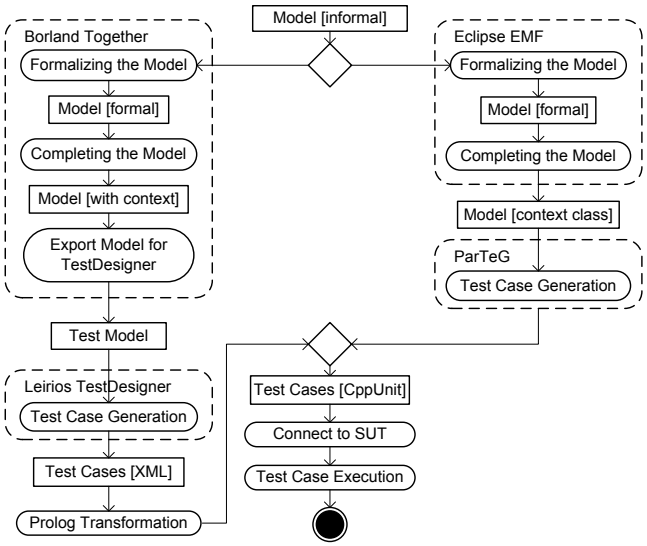


Fig. 1. The two toolchains.

TestDesigner [3] from Leirios (the company name has been changed to Smartesting) and generated abstract test cases. TestDesigner creates test cases in the form of XML documents. We used the Prolog transformation program to transform these XML files to CppUnit [4] tests.

3.2 Research Tool ParTeG

As an alternative to using the industrial toolchain, we also used and adapted the free model-based test generation tool ParTeG [5], which is based on the Eclipse Modeling Framework [6]. The input models for ParTeG are UML state machines in the context of UML classes that are both modeled using the UML 2.1 plugins [7]. Possible output formats are JUnit 3.8 and 4.3 [8].

We had access to the sources of ParTeG. Thus, we see the advantages of using ParTeG in the possibility of adapting the necessary test output format and in implementing unexpected features and interpretations of the system model. An anticipated disadvantage of using ParTeG was its prototype-related immaturity.

4 Adaptation and Formalization of the Model

In this section, we describe the necessary model adaptations and formalizations to automate the test case creation. The original system models were provided by the engineers at Thales. The formalization using OCL/UML was done in cooperation with the engineers at the Fraunhofer FIRST based on the requirements. The system models consist of four UML state machines, which describe the communication behavior of several train modules. There is one main machine that

references the other three. These referenced state machines are subcontrollers that describe the failure handling of the train modules. All of these system models have several flaws that are caused by insufficient formalization. In the following, we describe these flaws and how we removed them.

4.1 Formalization

Here, we present the individual steps to formalize the given models.

Removing Syntactical Errors. The first thing we discovered is a violation of the UML syntax: Outgoing transitions of the system model's initial states contain triggers although the UML specifies that such transitions must not have triggers. We solved this violation by transforming the initial state into a state named *Initializing* and creating a new initial state that is connected to the state Initializing via an additional transition.

After this transformation, the model was syntactically correct. However, state information is used in the test oracle and the SUT has no state called Initializing. Thus, the model does not represent the SUT's behavior anymore, and every generated test case would fail. As a solution, we removed all test steps that check for the state Initializing.

Observing Finalization Behavior in Subcontrollers. Observing the successful termination of a *subcontroller* and returning a corresponding verdict is important for test generation. Thus, it was necessary for us to observe states whose outgoing transitions lead to a subcontroller's final state. We call the states with the outgoing transitions of interest *Finalize* states. The original models contained untriggered completion transitions to model these state changes. Consequently, the test adapter was not able to trigger the entry of the final state.

As a solution, we added a trigger *notification* to the completion transitions. The result of this is that the test adapter could explicitly trigger these transitions to reach the final state. Since this new trigger is not part of the SUT's behavior, this solution also needs adaptation of the test adapter.

Flattening Hierarchical State Machines. One effect of the previous model transformation is that outgoing transitions of a subcontroller may lead to leaving the subcontroller while a *Finalize* state is active. This behavior is not intended by the Thales engineers. Instead, the SUT has to finalize and terminate the subcontroller after a *Finalize* state has been reached. A model transformation for compensating for this unintended effect consists of creating explicit outgoing transitions for all states of the subcontroller but the *Finalize* states. This corresponds to flattening a part of the system model. Since this introduces additional elements in the model, it increases the coverage of the SUT [9].

Formalizing Conditions. The guard expressions in the model contain no typed variables or constants. Instead, expressions are written in an informal style like prose. We derived a list of all used identifiers (variables and constants). The engineers at Thales provided the corresponding C++ data types and initial values for them. With this type information, we added formalized expressions to the system model using the Object Constraint Language [10].

4.2 Adding A Context

Based on the formal identifiers of the previous formalization step, we created a structural context for the state machine. This context consists of a class diagram and an optional object diagram. TestDesigner needs both of them. For ParTeG, providing the class diagram for the state machine is sufficient for test generation.

In the context class, we defined all formal identifiers as class attributes. In most cases, the mapping of simple C++ data types into the class diagram was straightforward. As an exception, we had to map unsigned integers to the UML data type *integer* and constrain it in a later step to non-negative values. The context class also contains setter methods for changing values of class attributes and operations to map triggers from the state machine to the test adapter.

The object diagram represents an instance of the context class. For automatic test generation with TestDesigner, the object diagram defines the initial system attribute value assignment.

4.3 Toolchain-Specific Model Transformations

In this subsection, we present toolchain-specific model transformations that were used to overcome restrictions of modeling tools and to keep the compatibility of test cases and the test adapter that is already in use at Thales.

Disjoint Triggers. First, we experienced problems with transitions that contain two or more triggers. In contrast to the UML standard, Together is not able to create transitions with two or more triggers. We solved this problem by splitting the transition into two or more parallel transitions, each handling a single trigger.

This transformation preserves the semantics, but changes the structure. This has an impact on the generated test suite. For instance, the satisfaction of All-Transitions on a system model with split transitions forces the test generator to traverse more transitions and, thus, to create larger test suites. Likewise, this also has an impact on the fault detection capability of the generated test suite [9].

Timed Triggers. Prior to our project, the engineers at Thales established an interface for the test driver to control system time. Due to restrictions of the test generation tools, we use function call events instead of using standard UML time events. A unique name scheme enables the test adapter to map these function calls to time information of the SUT.

Output Format Transformation. The Thales test framework, for which we had to generate test cases, requires test cases to be written in CppUnit. TestDesigner, as a general purpose test case generator, generates XML files but no CppUnit test files. Thus, we provided a text transformation program based on Prolog to convert the XML files into CppUnit files. After this transformation, the test cases from TestDesigner are executable in the test framework.

We integrated CppUnit test code generation for the Thales test framework directly into ParTeG. Thus, the test suite created by ParTeG did not need any further transformation.

5 Test Adaptation

The following section describes the used test adapter. We used the system model for automatic test generation. Since the system model is comparatively close to the implementation, the gap between the abstraction level of the model and the implementation is likewise small. Nevertheless, adaptation is required to execute the abstract test cases. There are several approaches to implement this adaptation, such as the concretion of the test cases by a model-to-text transformation or the use of an additional test adapter that maps from system models to SUT. We used a mixed approach [11, page 285] to bridge the gap.

The corresponding test adapter defines a test interface that is used to execute the partly transformed test cases. It transforms abstract trigger information of the test cases into concrete events and function calls that are forwarded to the controller within the test framework. Furthermore, information about the states of the system model are not explicitly present in the implementation and the test adapter maps them to system attributes in order to check state invariants of the system model. In general, we simplified complex data and represented them in an abstract way in the system model according to the recommendations for building test models by Utting and Legeard [11]. The task of the test adapter was to reinsert this complexity in the test cases.

6 Evaluation

In this section, we evaluate the different approaches to test case creation by comparing the code coverage and the size of the corresponding test suites. Line and branch coverage of the tests are demanded by the certification authorities. Other measures for a test suite's quality are mentioned in Section 8.

In Table 1, we describe four different test suites: the manually created test suite, the test suite generated by TestDesigner to satisfy All-Transitions, and two test suites generated by ParTeG. The first ParTeG test suite (*ParTeG 1*) just satisfies Multiple Condition Coverage on the system model, whereas the second one (*ParTeG 2*) additionally satisfies Multi-Dimensional [12] and contains sneak path analysis and model transformations like flattening the model or splitting choice pseudostates [9].

Test Suite	Line Coverage	Branch Coverage	Number of Test Cases
Manually Created	92.98 %	92.86 %	26
TestDesigner	87.19 %	83.33 %	141
ParTeG 1	91.32 %	91.67 %	252
ParTeG 2	94.63 %	92.86 %	2280

Table 1. Code coverage and size of the test suites.

Both test generation tools were restricted to use the given system models for test generation. In contrast, the human testers were able to use the models and also additional information like the source code of the controller or abstract information about the system environment like railroad track information. Correspondingly, the achieved coverage of the manually created test suite is higher than most of the automatically generated test suites. The only exception is the test suite for *ParTeG 2*: It is several times larger than the manually created test suite but covers a higher percentage of the source code. Since ParTeG 2 was generated automatically from the system model, the costs for test generation of this comparatively large test suite are neglectable.

None of the existing test suites covered 100% of lines or branches. The major reason for this is that some of the required test information, such as a model of a railroad track, are not included in the system model and, thus, could not be used for testing.

Since the SUT is already in use, the main objective of our project was not to detect undetected failures, but to improve the existing test process using MBT technologies. Thus, we compared the failure detection capabilities of the test suites using code coverage. Reasons for detected differences were found in the models and requirements.

7 Results and Lessons Learned

In this section, we present the results and lessons learned during the pilot project.

Before the start of the project, the engineers at Thales used models as pictures to support system development. Using these pictures, system designers could communicate their designs to the company's developers. Both knew about the informal style of the models and communicated directly with each other when something was unclear. Since the designers do not have to take care of precise syntax and semantics, this type of imprecise modeling is easier than designing formal models. For automatic test generation, however, precise and formal models are needed. As presented in the following, creating these formal models for automatic test generation caused more problems than expected.

First, as the applied transformations show, the designers of the system models interpreted the UML in a different way than the test tools do. This kind of semantic error was much harder to fix than the syntax errors. The reason for this is that removing semantic errors needed a detailed investigation and a higher

cooperation effort. Most time was spent on understanding *why* the design is wrong and *how* to correct it.

Second, some of the generated test cases are not reasonable. This was caused by missing environment information in the system models. We created no model of the environment and the test adapter did not check for a corresponding consistency of the test data. A solution to this problem is to provide a model of the environment to the test case generators, e.g. by adding information about railroad tracks like turnouts or the current train position.

Furthermore, we consider the repeatability of our actions. The concrete actions for removing syntactic and semantic issues cannot be reused in other projects or on other models because they differ from case to case. For instance, guidelines for designing models may vary for each project. The automatic transformations for adapting the test cases to the test adapter, however, can be repeated. Some transformations (see [9]) are applied to models and can be performed automatically. Transformations like the presented test design tool- and test adapter-specific ones can also be automatically reused in other projects.

Part of our evaluation was also the comparison of manually created test suites and automatically created ones. The large number of test cases generated by ParTeG tests the SUT extensively. One drawback is the execution time of some of the automatically generated test suites. ParTeG 2 designed roughly a hundred times more test cases than the human test designer, resulting in an increased execution time. However, the larger ones of the automatically generated test suites also covered a higher percentage of the SUT than the manually created test suite, and the test design is done automatically.

We also compared the two applied test generation tools. ParTeG generated the test cases in less than ten seconds. TestDesigner needed 25 minutes to generate a test suite. ParTeG reached at least the same or even a higher code coverage than the manual test cases when the strongest generation criteria (ParTeG 2) are applied. In general, test generation is undecidable and each applied test generation algorithm fits only to certain kinds of models. Thus, this is no general comparison of both tools but only an additional measurement of our project.

In retrospective, we encountered many unforeseen obstacles. Although we knew that MBT is not applicable "out-of-the-box" and we were prepared to customize our toolchains, we were surprised by the number of issues. Using even the latest version of the tools did help reducing costs, e.g. for creating the additional test adapter. On the other side, the use of automatic test design also helped saving costs. In contrast to one person week for manually updating the test suites, automatic test generation requires only a few minutes for updating the model and generating the test suite again.

8 Related Work

Several books provide surveys of conventional testing [13–15] and model-based testing [11, 16]. Many modeling languages have been used to create system models. The UML [17] is a popular representative that has been used by many authors

to demonstrate test generation techniques [18, 19]. In this paper, we used UML state machines.

Complete testing of all aspects is usually impossible – especially for reactive systems. Coverage criteria are widely adopted means to measure test suite quality. There are many kinds of coverage criteria (e.g. focussed on data flow or control flow) [11, 20]. Test generation can be stopped if a selected coverage criterion is satisfied. During our cooperation, we used different structural coverage criteria on UML state machines as a test generation stopping criterion.

There are several publications that present experience reports of model-based testing. For instance, Pretschner et al. [21] present an experience report on model-based testing. They include many aspects of model-based testing such as comparing coverage with error detection and model coverage with code coverage. In [11], Utting and Legeard present several reports on model-based testing. Their focus, however, is on the test generation technique and not on the acceptance thresholds when introducing model-based testing as a new testing technique.

There are many commercial model-based test generators for UML state machines available. For instance, the Smartesting Test Designer [3] supports the satisfaction of All-Transitions. Rhapsody ATG [22] is capable of creating test suites to satisfy MC/DC. Further commercial test generators are listed and compared in [23]. In this report, we applied the free test generation tool ParTeG [5] and the commercial test generator TestDesigner [3].

9 Summary

In this paper, we reported on our efforts to introduce model-based testing as a new testing technique in an industrial context. The results of the presented pilot project are that the introduction of MBT causes costs in the beginning. After establishing the necessary basis, however, MBT provides many advantages like automatic test design or reduced maintenance costs by fast response to requirements changes. Finally, we experienced that the customer's main point of interest for applying MBT is not the set of features (e.g., supported coverage criteria) provided by the test generator, but integrating MBT in the test process at all. Thus, it seems to us like the industry may already be aware of the possible benefits of MBT but fears the issues and costs of its integration.

References

1. "Norm DIN EN50128, Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems." CENELEC, 2001.
2. Borland, "Together," Januar 2010, http://www.borland.com/de/products/together/index.html.
3. Smartesting, "Test Designer," http://www.smartesting.com.
4. Sourceforge, "CppUnit 1.12 – Unit Tests for C++," http://sourceforge.net/projects/cppunit, 2008.

5. S. Weißleder, "ParTeG (Partition Test Generator)," May 2009, http://www.parteg.sourceforge.net.
6. Object Management Group, "Eclipse Modeling Framework (EMF)," May 2009, http://www.eclipse.org/modeling/emf/.
7. Eclipse, "Model Development Tools (MDT) - UML2," www.eclipse.org/uml2/, 2007.
8. K. B. Erich Gamma, "JUnit 4.1 - A Testing Framework for Java," http://www.junit.org, 2006.
9. S. Weißleder, "Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation," in *MoDELS*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds., vol. 5795. Springer, 2009, pp. 211–225.
10. Object Management Group, "Object Constraint Language (OCL), version 2.0," http://www.uml.org, 2005.
11. M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.
12. N. Kosmatov, B. Legeard, F. Peureux, and M. Utting, "Boundary coverage criteria for test generation from formal models," in *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 139–150.
13. P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 2008.
14. R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
15. G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.
16. M. Broy, B. Jonsson, and J. P. Katoen, *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer, August 2005.
17. Object Management Group, "Unified Modeling Language (UML), version 2.1," http://www.uml.org, 2007.
18. J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," in *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, R. France and B. Rumpe, Eds., vol. 1723. Springer, 1999, pp. 416–429.
19. M. Friske and H. Schlingloff, "Improving Test Coverage for UML State Machines Using Transition Instrumentation." in *SAFECOMP'07: The International Conference on Computer Safety, Reliability and Security*, ser. Lecture Notes in Computer Science, F. Saglietti and N. Oster, Eds., vol. 4680. Springer, 2007, pp. 301–314.
20. J. J. Chilenski, "MCDC Forms (Unique-Cause, Masking) versus Error Sensitivity," in *white paper submitted to NASA Langley Research Center under contract NAS1-20341*, January 2001.
21. A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One Evaluation of Model-Based Testing and Its Automation." in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 392–401.
22. IBM (Telelogic), "Rhapsody Automated Test Generation," http://www.telelogic.com/products/rhapsody.
23. C. J. Budnik, R. Subramanyan, and M. Vieira, "Peer-to-Peer Comparison of Model-Based Test Tools." in *GI Jahrestagung (1)*, ser. Lecture Notes in Informatics, H.-G. Hegering, A. Lehmann, H. J. Ohlbach, and C. Scheideler, Eds., vol. 133. GI, 2008, pp. 223–226.

Including Model-Based Statistical Testing in the MATERA Approach

Andreas Bäcklund, Fredrik Abbors, and Dragos Truscan

Åbo Akademi University, IT Dept., Joukahaisenkatu 3-5B, 20520, Turku, Finland
Andreas.C.Backlund@abo.fi, Fredrik.Abbors@abo.fi,
Dragos.Truscan@abo.fi

Abstract. In this paper, we present a Model-Based Testing (MBT) approach in which statistical data contained in Unified Modeling Language (UML) models are used to prioritize test cases. The models are used by a test derivation tool for automatic generation of test cases. The statistical data included in the models is used by the tool to determine the order of the resulting test cases before being implemented and executed. The test outputs are analyzed and information about requirement coverage is gathered. Based on the gathered statistics, the results are automatically fed back to the UML models to prioritize those sections of the system where failures are frequent.

1 Introduction

The complexity of software systems is constantly increasing. Hence, the amount of tests needed to properly test a software system is also increasing. Software companies usually do not have enough time to run all their test cases, and are therefore forced to prioritize them in such a way that the test cases cover as much functionality of the system as possible [1].

Especially in the telecommunications domain, which we target in this paper, the amount of test cases needed to be executed against the System Under Test (SUT) is rather large, and in practice only a part of these tests can be executed. Thus, there is a need to be able to order the test cases based on their importance. By determining the priority-specific paths within the system, it is possible to order the test cases in such a way that test cases of statistically higher priority are executed before others. In this way, specific sections of the system can be given higher priority, resulting in earlier execution of test cases running the highest prioritized paths of the system.

There are several benefits with using statistical testing [2, 3]. One of the main benefits is that more testing effort can be put into the most important sections of SUT, while less important section can be left less tested. Another benefit of conducting statistical testing is that statistical data from previous iterations of the testing process can be included in latter iterations, in order to target the test execution towards the system sections that are more important or yielded more failures.

Model-Based Testing (MBT) [4] is a testing approach that addresses some of the shortcomings in traditional testing by using an abstract representation (a *model*) of the system for automatic generation of test cases. The models can be implemented either as program code representations or as graphical representations using graphical specification languages, such as the Unified Modeling Language (UML) [5] or various tool

specific languages. The main idea with MBT techniques is to automatically generate tests by applying algorithms that are able to explore paths through the model.

According to [1], statistical testing can be integrated into the development process at the point when requirements have been gathered and approved. In other words, statistical testing can be initialized at the same phase as the model construction in MBT. Combining this with the benefits of using models to prioritize certain sections of the SUT, makes statistical testing beneficial when used in a MBT process.

There are several advantages of using MBT in a software development process. One advantage is that large amounts of tests can be generated in a short amount of time when there exists an appropriate model representation of the system. This adds additional value especially to conducting regression testing in the end of the software development project. Another advantage is that models are usually easier to modify than manually created test cases, which especially benefits projects where requirements are changing frequently. The third advantage is that the modeling of the system can be initiated immediately when the requirements have been specified. This means that a testing process using MBT can already be initiated in the design phase. Since the test model in MBT is typically an abstract representation of the system, it is easier to maintain it compared to manually written test cases.

2 Related Work

Previous research on combining statistical testing and MBT has been done under the acronym Model-based Statistical Testing (MBST). For instance, Prowell [6] presents an approach in which the transitions of a test (usage) model are annotated with probability of occurrence information that is later used during test generation by the JUMBL tool. A similar approach, targeted at telecommunication protocols, is presented in [7]. An operational profile (a Markov process) is used to describe the usage and behavior of the SUT. The probabilities included in the operational profile are later on used during test generation. In our approach we will use a test model describing the behavior of the system. The generated test cases will be ordered *after* test generation based on the statistical information, and information resulted from test reporting will be used to update the priorities for the generated test cases. In addition, requirements of the system are modeled and traced throughout the testing process.

Other similar work on MBST is presented in [8–10]. For instance, the author of [8] uses UML activity diagrams to express high level requirements. The nodes and edges in the activity diagram are assigned with weights indicating priority, based on complexity and possibility of occurrence of defects. The activity diagram is later translated into a tree structure, from which prioritized test scenarios are generated.

Work related to statistical testing has also been preformed in the context of the MaTeLo tool [11, 12]. In MaTeLo, test cases are generated from statistical models of the SUT expressed using Markov chains usage models. However, while MaTeLo-based approaches utilize a usage model for describing the SUT, our approach utilizes a system model to represent the SUT.

In [9] the author presents an approach for using MBST together with time durations to test real-time embedded systems. The author's approach differs slightly from ours, since it uses statistical information to test the reliability of the system. In the approach,

reliability is tested by generating test cases from a model that represents the actual use of the system. In our approach, statistical information about the system is not used to test the intended usage of the system, but rather to order test cases according to weighted probabilities calculated from statistics of requirement priority and use case probability.

The most similar approach is presented in [10]. Here the authors take advantage of an approach in which they go from a requirements document, via a statistical model, to a statistical test report. Similarly to our approach, their approach benefits from a high degree of automation in each phase of the testing process.

3 Overview of MATERA

MATERA (Modeling for Automated TEst deRivation at Åbo Akademi) [13] is an approach for integrating modeling in UML and requirement traceability across a custom MBT process (see Figure 1). UML models are created from the system requirements, using a UML modeling tool. The models are validated by checking that they are consistent and that all the information required by the modeling process is included. Consequently, the models are transformed into input for the test derivation tool. The resulting test cases are executed (after being concretized) using a test execution framework. The results of the test execution are analyzed and a report is generated. Requirements are linked to artifacts at different levels of the testing process and finally attached to the generated test cases. The approach enables requirements to be backtraced to models in order to identify which test cases have covered different modeling artifacts or from which part of the models a failed test case has originated.

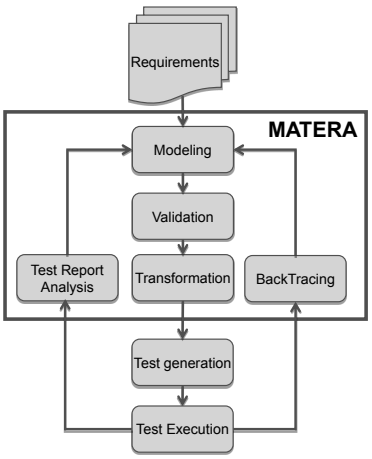


Fig. 1. MATERA process

4 Statistical Approach for MATERA

Our statistical approach relies on two sources of information: (1) that the functionality of the system (use cases) has associated *probability* values, depicting the chances for functionality to be invoked by the external user of the system during the use of the SUT; (2) that the requirements of the system are classified based on their importance (for testing) by associating them with *priority* values. The priorities and probabilities of the system are considered to be given from external sources (e.g., system requirements or stakeholder recommendations) and *a priori* to the first iteration of the testing process. In latter test cycles, the priorities can be adjusted based on statistics of uncovered requirements from previous test cycles for targeting the testing process towards a certain part of the SUT.

There is a slight difference between probability and priority. Even though they both mean that specific sections of the SUT are prioritized, it is important to recognize that probability is part of the model, while requirement priority is a property for ordering system requirements according to importance. Hence, UML use case elements are given a probability value indicating the chance of the use case to be executed, whereas requirements are given a priority value indicating their importance for testing. The values are manually assigned to each use case in part. The two types of values are then combined in the test model from where test cases are generated. Each resulting test case will have a *weighted priority* calculated based on the cumulative probabilities and priorities of the test path in the model. The weighted priority will be used for determining the test execution order. In the following, we delve into more details related to each phase of the process.

4.1 Requirements Modeling

The process starts with the analysis and structuring of the informal requirements into a Requirements Model. The requirements diagrams of the Systems Modeling Language (SysML) [14] are used for this purpose. Requirements are organized hierarchically in a tree-like structure, starting from top-level abstract requirements down to concrete testable requirements. Each requirement element contains a *name* field which specifies the name of the requirement, an *id* field, and a *text* field. For the purpose of statistical testing, requirements are also given a `priority` value (see Figure 2). The priority value is a property describing the importance of the requirement. During the modeling process the requirements are traced to different parts of the models to point out how each requirement is addressed by the models. By doing this we ensure the traceability of requirements and that priority information is propagated to other model artifacts.

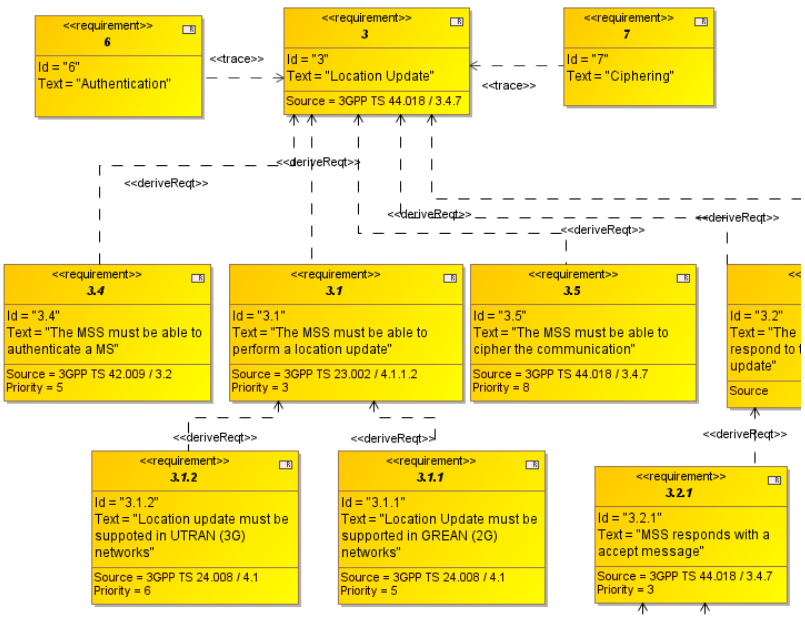


Fig. 2. Requirement Diagram with priorities

4.2 System Modeling

In this phase, the SUT is specified using UML. In our modeling process, we consider that several perspectives of the SUT are required in order to enable a successful test derivation process later on. A *use case diagram* is used to capture the main functionality of the system. *Sequence diagrams* are used to show how the system communicates with external components (in terms of sequence of messages) when carrying out different functionality described in the use case diagram. A class diagram is used to specify a *domain model* showing what domain components exist and how they are interrelated through interfaces. A *behavioral model* describes the behavior of the system using state machines. *Data models* are used to describe the message types exchanged between different domain components. Finally, *domain configuration models* are used to represent specific test configurations using object diagrams. Each use case is given a probability value which indicates the chance of the use case being executed (see Figure 3).

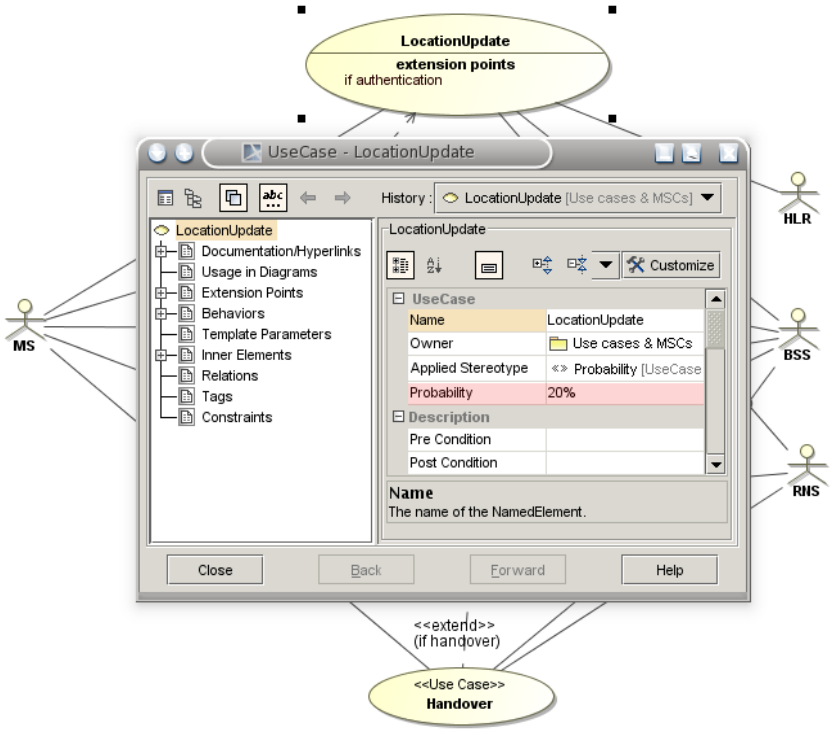


Fig. 3. Use case diagram with probability

The state model describing the expected behavior of the system is the pivotal artifact for test generation. According to the MATERA approach, leaf requirements are linked to transitions in the state machine to enable requirements traceability and requirements coverage during test generation. Thus, the priority of each requirement will be asso-

ciated to the corresponding transition. Similarly, use case probabilities are manually linked to the state model, as use cases are related with one or several starting points in the state machine diagram (see Figure 4). This enables the test generation tool to determine the weighted probability of certain paths through the state model. Before the tests are generated, the consistency of the UML models is checked using custom defined Object Constraint Language (OCL) rules [15].

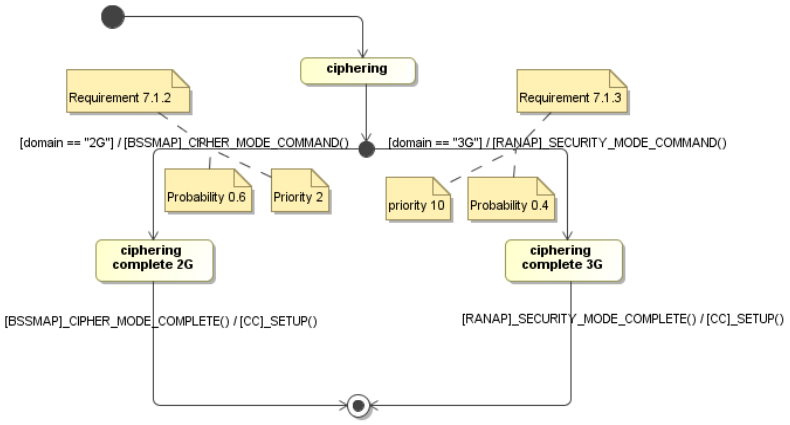


Fig. 4. UML state machine diagram

4.3 Test Case Generation

In the MATERA approach, the UML models are translated into a representation understood by a test generation tool, namely Qtronic [16], using the transformation described in [17]. During the translation, the priority and probability values are propagated to the new model representation. Test cases are generated by the tool based on the selected structural coverage criteria (e.g., state, transition, and requirement coverage, respectively), without taking into account priority and probability annotations.

4.4 Test Case Ordering

After the test cases have been generated, the test generation tool can determine the generation order of test cases based on the annotated probability and priority values. For each generated test case, a weighted probability is calculated based on the algorithm implemented by the test generation tool described in [18]. The weighted probability is calculated from both the use case probability and the requirement priority and determines the sequence in which test cases are ordered (see Figure 6). Test cases are finally rendered into executable test scripts using an adapter for concertizing test cases into executable scripts.

4.5 Test Execution

Test scripts are executed against the SUT using a test executor tool. The test scripts are executed in the order determined by the test generation tool. If only a part of the test suite can be executed, e.g. due to restricted testing time, ordering tests according to probability and priority ensures that the most important tests are executed. The execution of test scripts is monitored and the results are stored in log files. The log files contain information about the test execution, e.g. messages sent and received by the SUT, tested and untested requirements, used resources, etc. The log files together with the test scripts serve as a source for the test results analysis.

4.6 Test Log Analysis

By parsing logs and scripts and comparing these against each other it is possible extract statistical data from the test run. The extracted data describe requirements that have been successfully tested, requirements that have been left uncovered, and during testing of which requirements that failures have occurred.

The analysis of the test execution is presented in a HTML report (see Figure 5) generated by the MATERA tool-set. The report consists of two sections, one for General Test Execution Statistics and one for Requirements Information. The General Test Executions Statistics section contains information about the number of test cases that passed and failed. The Requirements Information section contains information about the requirement coverage. Finally, the test cases are presented in a Traceability Matrix.

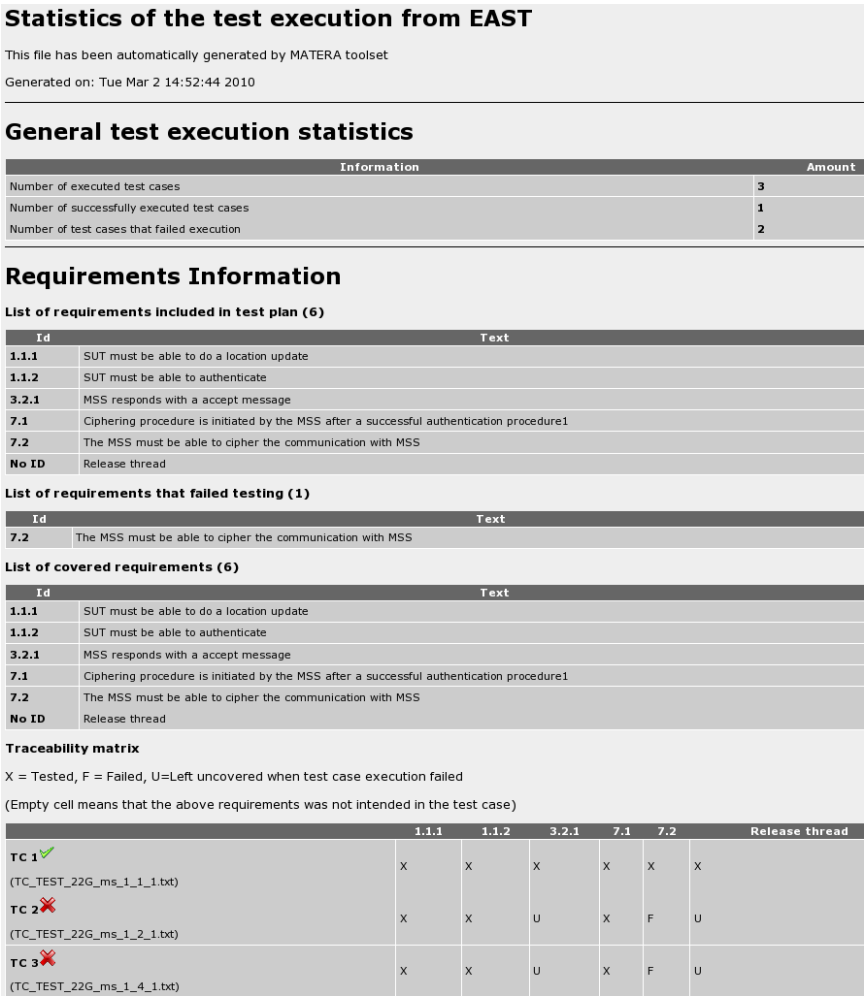
4.7 Feedback Loop

In the feedback loop, the statistical information gathered in the test log analysis is used to update priority of requirements that failed or were left uncovered during testing. The feedback loop is implemented as a part of the MATERA tool-set and allows the modeler to read in the analyzed statistics and update priority values for requirements in the UML models without user intervention.

The feedback loop is the main actor for targeting the test execution towards the parts of the system that had most failures. This is done by incrementally increasing the priority of the failed and uncovered requirements, such that they will counterbalance the effect that the probabilities of the use cases have on the ordering of tests. As testing progresses and the process is iterated several times, the importance (priority) of requirements will change according to how well they have been tested. Providing a feedback loop which updates the requirement importance automatically, will result in that the failed and uncovered requirements are included in the test cases that are ordered first in the test execution queue.

However, if requirement importance is changed due to external factors that cannot be derived from statistics, the tester can choose to manually change the priority of requirements directly in the models at any time.

The feedback module is executed from the MATERA menu in MagicDraw. When initialized, the module collects test data from a user specified folder holding test logs and test scripts from the last test execution. Based on these statistics, the priority values for requirements that need to be tested more thoroughly in a subsequent test iteration are



Statistics of the test execution from EAST

This file has been automatically generated by MATERA toolset

Generated on: Tue Mar 2 14:52:44 2010

General test execution statistics

Information	Amount
Number of executed test cases	3
Number of successfully executed test cases	1
Number of test cases that failed execution	2

Requirements Information

List of requirements included in test plan (6)

Id	Text
1.1.1	SUT must be able to do a location update
1.1.2	SUT must be able to authenticate
3.2.1	MSS responds with a accept message
7.1	Ciphering procedure is initiated by the MSS after a successful authentication procedure1
7.2	The MSS must be able to cipher the communication with MSS
No ID	Release thread

List of requirements that failed testing (1)

Id	Text
7.2	The MSS must be able to cipher the communication with MSS

List of covered requirements (6)

Id	Text
1.1.1	SUT must be able to do a location update
1.1.2	SUT must be able to authenticate
3.2.1	MSS responds with a accept message
7.1	Ciphering procedure is initiated by the MSS after a successful authentication procedure1
7.2	The MSS must be able to cipher the communication with MSS
No ID	Release thread

Traceability matrix

X = Tested, F = Failed, U=Left uncovered when test case execution failed

(Empty cell means that the above requirements was not intended in the test case)

	1.1.1	1.1.2	3.2.1	7.1	7.2	Release thread
TC 1 (TC_TEST_22G_ms_1_1_1.txt)	X	X	X	X	X	X
TC 2 (TC_TEST_22G_ms_1_2_1.txt)	X	X	U	X	F	U
TC 3 (TC_TEST_22G_ms_1_4_1.txt)	X	X	U	X	F	U

Fig. 5. Statistical Report

incremented with a predefined coefficient and automatically updated in the requirement models.

5 Tool Support

In our current approach we use No Magic's MagicDraw [19] modeling tool for creating and validating the UML models. The Graphical User Interface (GUI) of the MATERA tool-set has been implemented as a plug-in for MagicDraw. The purpose of the MATERA tool-set is to extend the capabilities of MagicDraw for specifying system models and using them as input for automatic test generation.

For automatic test case generation we use Conformiq's Qtronic [16]. Qtronic is an Eclipse based tool to automate the design of functional tests. Qtronic generates tests and

executable test scripts from abstract system models based on selected coverage criteria. An example of a test case sequence ordered by probability is shown in Figure 6. The models for Qtronic are expressed using the Qtronic Modeling Language (QML). QML is a mixture of UML State Machines and a super set of Java, used as action language. The UML state machines are used to describe the behavior of the SUT and QML is used to represent data and coordinate the test generation. By using a custom Scripting Backend (adapter), Qtronic generates executable test scripts for the Nethawk's EAST test executor framework [20].

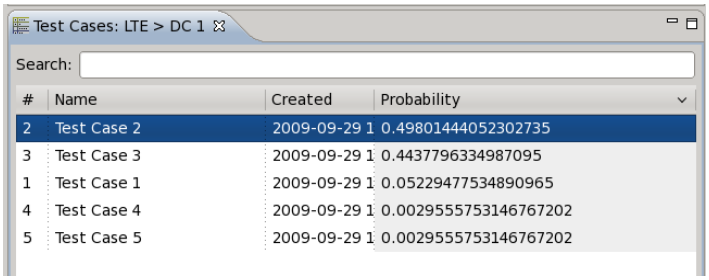


Fig. 6. Test case sequence ordered by weighted probability in Qtronic

The EAST Scripting Backend in Qtronic is the main actor for rendering the test scripts. When the abstract test cases are selected for execution, they are rendered to test scripts, loaded into the EAST test executor, and executed against the SUT. The test executor produces logs from the test case execution, which are used as source for the statistical analysis in the MATERA tool-set.

6 Conclusions

In this paper, we have presented a model-based testing approach in which statistical information is included in the system models and used for ordering of test cases. The approach benefits from a highly integrated tool chain and a high degree of automation. To handle complexity, the system is described from different perspectives using a different UML model for each perspective. Statistical information is described in use case and requirement diagrams, via priority and probability annotations. Traceability of requirements is preserved in each step of the testing process and can be gathered as statistics for later test cycles.

During test generation, test cases are ordered based on the statistical information contained in the models. After each test run, statistical information is gathered and fed back to the models in a feedback loop. The statistical information serves as basis for updating the information contained in the models to prioritize tests for those parts of the system where failures are discovered.

Future work will be to extract additional information from test logs. Since the test logs contain detailed information about messages sent and received from the SUT, this information could be extracted and presented to the user. For example the HTML test

report could be extended to include sequence diagrams for each test case. The tester could then examine failed tests in more detail, e.g. see what messages has been sent and received and what values were used, to manually adjust priorities and probabilities in the model. It could also facilitate the debugging of possible errors in the model.

References

1. Weber, R.J.: Statistical Software Testing with Parallel Modeling: A Case Study, Los Alamitos, CA, USA, IEEE Computer Society (2004) 35–44
2. Mills, H.D., Poore, J.H.: Bringing Software Under Statistical Quality Control. Quality Progress (nov 1988) 52–56
3. Whittaker, J.A., Poore, J.H.: Markov analysis of software specifications. ACM Trans. Softw. Eng. Methodol. (1) (1993) 93–106
4. Utting, M., Pretschner, A., Legeard, B.: A Taxonomy of Model-Based Testing. Technical report (April 2006)
5. Object Management Group (OMG): OMG Unified Modeling Language (UML), Infrastructure, V2.1.2. Technical report (November 2007)
6. Prowell, S.J.: JUMBL: A Tool for Model-Based Statistical Testing. In: HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, Washington, DC, USA, IEEE Computer Society (2003)
7. Popovic, M., Basicevic, I., Velikic, I., Tatic, J.: A Model-Based Statistical Usage Testing of Communication Protocols. 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS) (2006) 377–386
8. P.G., S., Mohanty, H.: Prioritization of Scenarios Based on UML Activity Diagrams. First International Conference on Computational Intelligence, Communication Systems and Networks (2009) 271–276
9. Böhr, F.: Model Based Statistical Testing and Durations. In: 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, IEEE Computer Society's Conference Publishing Services (CPS) (March 2010) 344–351
10. Bauer, T., Bohr, F., Landmann, D., Beletski, T., Eschbach, R., Poore, J.: From Requirements to Statistical Testing of Embedded Systems. In: SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems, Washington, DC, USA, IEEE Computer Society (2007)
11. All4Tec: MaTeLo http://www.all4tec.net.
12. Dulz, W., Zhen, F.: MaTeLo - Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3. International Conference on Quality Software (2003) 336
13. Abbors, F., Bäcklund, A., Truscan, D.: MATERA - An Integrated Framework for Model-Based Testing. In: 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2010), IEEE Computer Society's Conference Publishing Services (CPS) (March 2010) 321–328
14. Object Management Group (OMG): Systems Modeling Language (SysML), Version 1.1. Technical report (November 2008)
15. Abbors, J.: Increasing the Quality of UML Models Used for Automatic Test Generation. Master's thesis, Åbo Akademi University (2009)
16. Conformiq: Conformiq Qtronic (2009) http://www.conformiq.com.
17. Abbors, F., Pääjärvi, T., Teittinen, R., Truscan, D., Lilius, J.: Transformational Support for Model-Based Testing–from UML to QML. Model-based Testing in Practice 55
18. Conformiq: Conformiq Qtronic User Manual. (2009) 131–134 http://www.conformiq.com/downloads/Qtronic2xManual.pdf.
19. No Magic Inc: No Magic Magicdraw (2009) http://www.magicdraw.com/.
20. Nethawk: Nethawk EAST test executor (2008) https://www.nethawk.fi/.

Test Case Generation for Product Lines based on Colored State Charts

Manal Farrag[1], Wolfgang Fengler[1], Detlef Streitferdt[2], Olga Fengler[1]

[1] Technical University of Ilmenau, Germany
{manal.farrag | wolfgang.fengler | olga.fengler}@tu-ilmenau.de
[2] ABB Corporate Research, Ladenburg, Germany
detlef.streitferdt@de.abb.com

Abstract In this paper a model-based, reuse-oriented test technique is presented, called Colored Model-Based Testing for Software Product Lines (CMBT-SWPL). It is a new requirements based system testing method used for validation and verification of product lines. A key concept is the Colored State Chart (CSC), which considers variability early in the product line development process. During domain engineering the CSC is derived from the usage model and the feature model. By coloring the State Chart the behavior of several product line variants can be modeled simultaneously in a single diagram. The CSC represents the domain test model, out of which test models are derived, in turn the input to the test case generation using statistical testing methods. During application engineering these colored test models are customized for a specific application of the product line. Finally the generated test cases are executed and the test results are ready for evaluation. In addition to test case generation, the CSC will be transformed to a Colored Petri Net (CPN) for verification and simulation purposes.

1 Introduction

"A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."[1]. The following figure 1 is the product line engineering reference process which we developed to compare the development artifacts in parallel to the test artifacts. This reference process was influenced by Klaus Pohl's [2] and Gomaa's [3] frameworks.

As depicted in figure 1 the product line engineering reference process consists of the domain engineering process and the application engineering process. In the domain engineering the core assets are built for all members of the product line. The core assets are the artifacts used in the development of the product line such as requirements, design, implementation and test artifacts. In application engineering the core assets are reused and customized to produce a specific application of the software product line. The reuse addressed here is strategic planned reuse that is targeted to minimize effort, cost, time and produce high

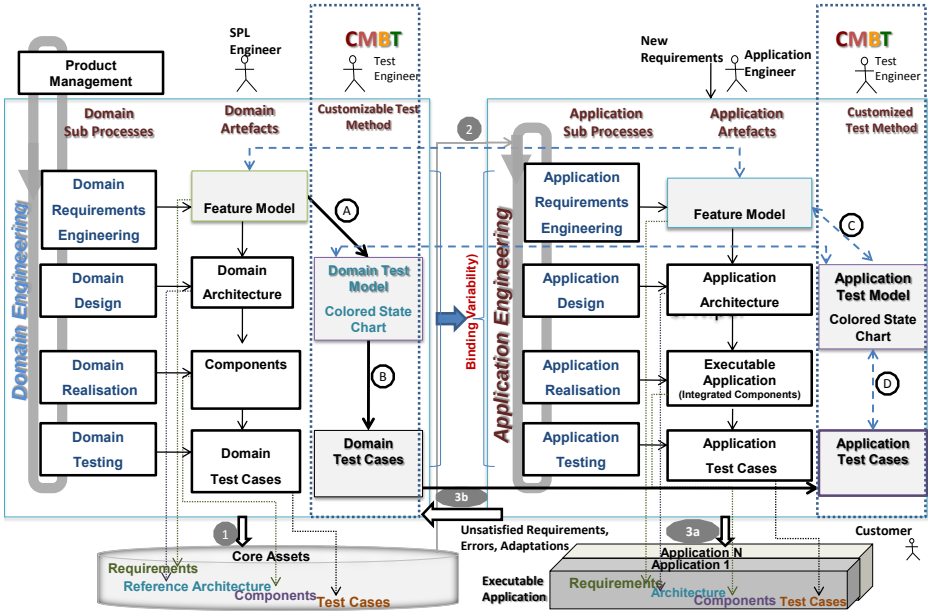


Figure 1: Colored Model-Based Testing for Software Product Lines(CMBT-SWPL) in the Product Line Engineering Reference Process

quality products. Variability is closely related to the concept of software reuse. The benefits of reuse were realized for all development artifacts but still not for testing. It is desired to create test artifacts that consider variability early in the development process, i.e. in the domain engineering phase in order to be reused in the application engineering phase.

The two main aspects that we like to focus on when testing product lines, are as we mentioned before (1) reuse and the second one which is important to achieve this type of predictive or strategic reusability is (2) model based testing.

This paper describes a new approach for validation and verification of product lines called the CMBT-SWPL. This requirements-based testing approach uses CSCs[4]. Requirements-based testing can increase efficiency, reduce the risk of failure, and improve overall software quality.[5] The behavior of various product variants can be modeled simultaneously in a single figure by coloring State Charts, and thus address product line variability early in the product line development life cycle.

The remainder of the paper is organized as follows. In section 2, a running example is provided in form of the Universal Remote Control. The relevant artifacts of this system are described. Section 3 represents the Colored State

Chart (CSC) starting by showing in 3.1 an example how the folding is performed and then in 3.2 an overall CSC is formally defined. Section 4 describes how the test models can be derived. First, the CSC used in the example is presented. Next, it is described how a simple State Chart can be derived from the CSC for a given feature set. Finally, the Statistical Testing approach is described and it is explained how test cases for different feature sets can be derived in this way. The paper ends with a summary and outlook.

2 Example: Universal Remote Control (URC)

The example of a URC is used to explain the test method. This example [6] was conducted at the Faculty of Computer Science and Automation - Institute of Computer Engineering - Ilmenau University of Technology and it is part of the Digital Video Project(DVP)[7][8] based on the VDR project [9]. The URC is modeled using features where a feature stands for "a logical unit of behaviour that is specified by a set of functional and quality requirements representing an aspect valuable to the customer and system architect" following the definitions in [10]. Feature models first described by Kang 1990 [11] model a complete product line by hierarchically organized features, which might be mandatory or optional. The set of all mandatory features composes the core of the product line, present in all derived applications. Each set of selected optional features (according to the given constraints, e.g. requirements) together with the mandatory core is used to derive an application of the product line.

The URC in [6] has the following features: (1) Controlling the video recorder (2) Controlling other devices and it should (3) provide a user profile. All the functionalities are explained in detail in [6]. In this paper we are going to concentrate only on how to control the video recorder. The features we are going to focus on are based on the overall feature model represented in [6] and are reduced to the feature model in figure 4. In the reduced feature model, there are mandatory features such as **Basic Functions**. The **Basic Functions** feature could contain other features such as **choosing channel** feature, **controlling volume** feature or **waiting** feature. The optional features for controlling the video recorder (VDR) are **Electronic Program Guide (EPG)** and Title-Database (which is out of our scope). The **Reminder** feature is available to remind a user of e.g. movies and is a sub feature of the EPG feature.

3 Colored State Charts

The State Charts (SCs), that are used here, are based on David Harel´s State Charts, which were introduced 1987 in [12]. UML State Charts are based on Harel's State Charts [13] with modifications like enhanced synchronisation properties or overlapping states. The basics of the State Charts are covered in [14], [13] as "UML state machine diagrams".

UML State Charts cannot address product lines. Thus, the extended State Chart version as introduced in [4] and [15], will be used in this article. Such State

Charts are called “Colored State Charts” (referred to as CSC) and are based on the basic idea of Colored Petri Nets (CPN), as described in [16] or [17].

The basic principle is based on the folding of similar state diagrams, which represent, for example, a class and its subclasses, or more object instances of a class. The example in figure 2 is depicted to explain the idea.

3.1 Example: Folding of States and Transitions

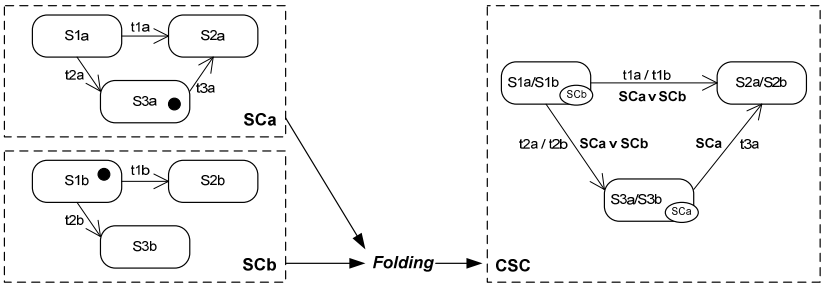


Figure 2: Folding of States and Transitions (from [4])

Here, State Charts SCa and SCb have the same states, but differ only in the transitions. SCa and SCb have the transitions t1a and t1b respectively as well as t2a and t2b. However, t3a exists only in SCa. The included tokens (black dots) in S3a and S1b show the currently active states in the state diagrams SCa and SCb. Tokens move when transitions fire into the next active state. In the resulting CSC, the common states and transitions are superimposed. This superimposition is labeled as shown in figure 2. For example, the two states S1a and S1b will be superimposed in the CSC to S1a/S1b and correspondingly the two transitions t1a and t1b are superimposed in the CSC to t1a/t1b. The states or transitions (e.g. t3a) that are only present in one state diagram are transferred to the CSC. In the CSC the tokens SCa and SCb appear in the states (S1a/S1b and S3a/S3b), as result of the superimposition of the corresponding marked states in SCa or SCb. The transitions of the CSC will be further labeled with the disjunction of the SC names, which are involved in the superimposition of the transitions. For example, the transition based on t1a and t1b will be labeled $SCa \vee SCb$ and the transition based on only t3a will be labeled SCa. The SCi names used in the tokens and on the transitions are described in the following sections as colors. Transitions can fire, if the originating state contains a token of color SCi (e.g. S1a/S1b contains the token SCb) and the disjunction of the transition contains SCi (e.g. the disjunction of t2a/t2b contains SCb).

3.2 Formal Definitions

The following CSC used in this article does not use all the options mentioned in [4] and [15]. In order to extend the State Chart (SC) to a Colored State Chart (CSC), based on the general State Chart definition, we are going to focus only on the following:

- S: a finite set of complex states. The elements of S are called s.
- T: a finite number of complex transitions with $T \subseteq S \times S$. The elements of T are resulting as (si, sj).
- C: a finite set of colors. The elements of C are called c.
- CV: a finite set of color variables. The elements of CV are called cv.
- m: a marking with m: $S \rightarrow P(C)$, were P(C) is the power set of C.
- ctf: a color transition function with $T \rightarrow CV$. For each $cv \in CV$ there is a definition set cvd with $cvd \subseteq C \wedge cvd \neq \emptyset$.

For the state transitions in the CSC, there is a transition firing rule:

- A transition (si, sj) can fire for $c \in C$ if $c \in m(si)$.
- A transition fires for $c \in C$ if and only if it can fire for c (and additionally the events and the guards of the transition are fulfilled).

The result of firing of (si,sj) for $c \in C$ is:

- $m^{k+1}(si) = m^k(si) \backslash \{c\}$ and $m^{k+1}(sj) = m^k(sj) \cup \{c\}$
- k is before firing (si, sj); k +1 is after firing (si, sj).
- A transition fires in parallel for different colors, if it can fire for these colors.

The CSC in figure 2 will be depicted in the overall following figure 3.

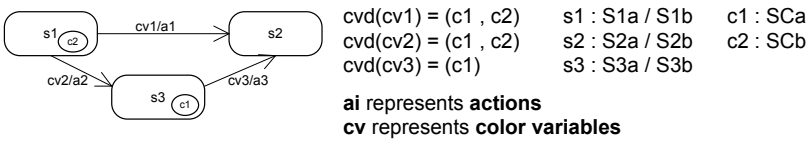


Figure 3: Example of a Colored State Chart(CSC)

Colored State Charts can be transformed to Colored Petri Nets [4]. This makes verification of the transformed model possible using widespread methods and tools developed for CPN.

4 Testing Product Lines with Colored State Charts

The traditional way of testing single systems is to test after coding is finished (for functions, components or the integrated system), i.e. when there is a running

application. However, in product line testing, we aim to start the test development early in the domain engineering phase and not to wait until the variability is bound and there is a running application. Therefore, testing a product line is divided into domain testing and application testing [2]. The targeted test here is a requirements model based system test that takes variability into consideration. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic[18].

4.1 CMBT: Domain Engineering.

Within the domain engineering (the left part of figure 1) of the product line a feature model was developed in [6] to model the variable and common features of the product line. For the further elaboration of the example we concentrate only on the features shown in figure 4, referenced as "Reduced Feature Model". The selection of features in the reduced feature model for a certain product line variant is called feature configuration. In parallel to the feature model a usage model is developed, represented as State Chart with usage probabilities. It models the black box usage of the system and is extended towards a Colored State Chart.

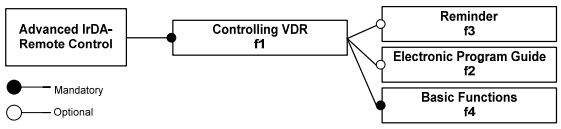


Figure 4: Reduced Feature Model

The reduced feature model results in three product line variants (V1, V2 and V3). V1 includes features f1 and f4. V2 includes features f1, f2 and f4. V3 includes features f1, f2, f3 and f4. The set of all features corresponds to the finite set of color variables of the colored State Chart. The features correspond to the color variables, presented in the test model in figure 5 (gray refers to feature f2, the Electronic Program Guide). This test model can be elaborated formally as explained in the following lines and based on Section 3.2:

- S represents the set of all states. We use symbolic identifiers for the elements of S. In our example: $S = \{Waiting,\ Volume,\ Channel,\ Electronic\ Program\ Guide(EPG),\ Reminder\}$
- T represents the set of all transitions. The elements of T are resulting from $S \times S$ as (symbolic identifier i, symbolic identifier j). In our example: $T = \{(Waiting, Volume), (Waiting, Waiting), \ldots\ldots\}$
- An example for a condition is: $condition(Waiting, Volume) = vol\_plus$
- An example for ctf is: $ctf(Waiting, Volume) = f4$

– The set of features is: $F = CV = \{f1, f2, f3, f4\}$
– The set of the defined product line variants is: $V = C = \{v1, v2, v3\}$
From the VD(v)in figure 5, the assignment of the product line variant vi to the feature fj can be directly derived as follows:
– $cvd(f1) = \{v1, v2, v3\}$
– $cvd(f2) = \{v2, v3\}$
– $cvd(f3) = \{v3\}$

Based on the knowledge captured in the feature model and the previously developed usage model a CSC is developed (see (A) in figure 1). The CSC includes the behavior of the system family and at the same time its variability, represented by the colors. At any given moment the CSC refers to one active product line variant while the remaining variants are passive. It is similar to the concept of instantiation in the object oriented paradigm. One active product variant is equivalent to a certain feature configuration extracted from the feature diagram or respectively from the CSC.

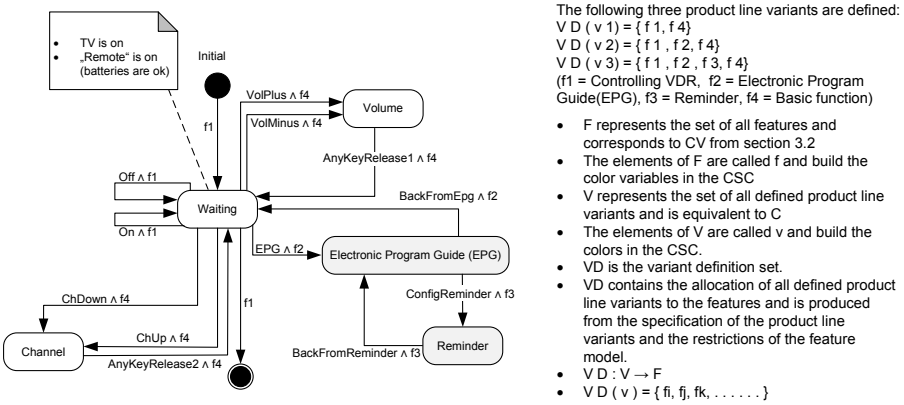


Figure 5: CSC-Test Model

One color, i.e. one product line variant may constitute of one or more features in this case color variables. Within the domain test model in figure 5 the variability is realized by mapping the features to one or more transitions. The events are combined with features present in a given application of the product line. A transition will only be able to fire if all features which are bound to it are present in the derived application.

The domain test model includes all features of the product line. Out of the domain test model, domain test cases (see (B) in figure 1) are derived by reducing the feature set to the common features, which is the core of the product line. Based on this reduction a State Chart is derived and enhanced with usage

probabilities to be used as input for the generation of test cases, described in section 4.3. The test artifacts that are gained until this step such as test models and test cases are stored in the repository to be reused for the derivation of applications of the product line.

4.2 CMBT: Application Engineering

Within the application engineering (the right part of figure 1) of the product line the feature model is reused. Based on the Application Requirements Engineering phase possible needed changes to the feature model are thoroughly assessed. In the case where such new requirements make changing the feature model worthwhile, these changes are fed back to the Domain Engineering phase. The next step is to customise the colored domain test model (i.e. one color is chosen) to produce the colored application test model for a specific application (see (C) in figure 1). The CSC is ransformed into a State Chart modeling the behavior of a single application and enhanced with usage probabilities to generate test cases for this application using the statistical testing approach described in section 4.3. Statistical testing as one of the technologies to reduce the huge test space was chosen based on own experiences in the embedded software development domain, expert knowledge of embedded software developers in the automation domain and the results of the D-MINT project. Other approaches towards test case reduction and generation are subject of further research.

4.3 Statistical Testing

Statistical testing [19], [20] is based on usage models. Such models represent the typical (based on statistics) usage of a system, in our case by an end user. The usage model may be expressed by a State Chart with annotated transition probabilities. All possible paths from the start to the final state form the test cases of the system.

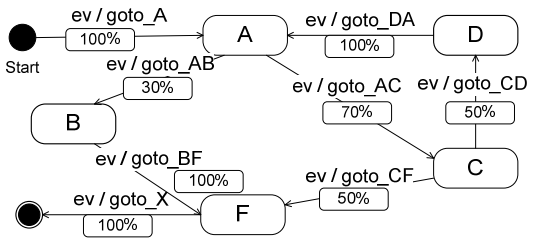


Figure 6: Statistical Testing based on State Charts

Each test case is composed of the transitions and states along a path from the start to the final state as depicted in figure 6. An example of a test case would be

the path with the transitions `goto_A`, `goto_AB`, `goto_BF` and `goto_X` in exactly this order. For each transition, test steps according to a test interface are defined, e.g. push buttons on the system under test (SUT) or the measurement of the reference values parallel to the test case execution. The probabilities attached to the transitions represent the typical usage of the system and allow the generation of an arbitrary number of test cases according to the statistical system usage.

The typical usage can be obtained by observing the user using the system, using similar project´s statistics or by expert estimations which can be further refined if necessary through observed usages. The tool JUMBL [21] is used to generate test cases according to the above mentioned criteria. The test model in 6 is represented as Graph Modeling Language (GML) and with minor manual adaptations transformed into The Model Language (TML) used as input for JUMBL.

5 Summary and Outlook

In this paper we presented a model-based, reuse-oriented test technique called the Colored Model-Based Testing for Software Product Lines (CMBT-SWPL). UML state machines do not consider product line variability. With the approach described in this paper, variability will be considered early by introducing it directly in the main product line components of the CSC. Thus, by using the CSC, the product line variability can be extended to UML state machines. One of the main benefits of the CMBT-SWPL method is its formal syntax as well as a formal semantic for the CSC and the variability. Currently the CMBT-SWPL method has been positively reviewed for its industrial feasibility, the future application of the method will deliver real measurements. The combination of applying the CMBT-SWPL method with the statistical testing is expected to lead to reduction of the testing efforts. As result of the D-MINT project [22] an overall improvement of 35% for the usage of model-based testing technologies in contrast to non-model-based testing was achieved (statistical testing was a central part of this project). The result is valid for development projects of single applications in the automation domain, product lines have not been targeted in D-MINT. Future research efforts will result in metrics on the improvement due to the CMBT-SWPL product line testing - we expect at least the improvement of the D-MINT project. Thus, the targeted strategic reuse, realised for the development artifacts, could be argued for the test artifacts as well. Last but not least, applying the CMBT-SWPL method enables the application of validation and verification techniques on the same model.

References

1. Clements, P., Northrop, L.M.: Software Product Lines : Practices and Patterns. 6th edn. Addison Wesley (2007)
2. Pohl, K., Böckle, G., Linden, F.v.d.: Software Product Line Engineering. Birkhäuser (2005)

3. Gomaa, H.: Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures. In Morisio, M., ed.: ICSR. Volume 4039 of Lecture Notes in Computer Science., Springer (2006) 440
4. Fengler, O., Fengler, W., Duridanova, V.: Modeling of Complex Automation Systems using Colored State Charts. In: ICRA, IEEE (2002) 1901–1906
5. MKS: Requirements-Based Testing: Encourage Collaboration Through Traceability, available online at http://www.softwaremag.com/pdfs/whitepapers/Requirements_Based_Testing.pdf?CFID=22695304&CFTOKEN=58434216; visited 2010-03-18, (2009)
6. Dietzel, R.: Konzeption und Entwicklung einer Systemfamilie für eine Universal-Fernbedienung auf Basis eines Palm-Handhelds. Diplomarbeit, TU-Ilmenau (2003)
7. Meffert, F.: Konzeption einer Videodatenverteilung im Rahmen des Digitalen Video Projektes (DVP). Diplomarbeit, TU-Ilmenau (2003)
8. Streitferdt, D.: Family-Oriented Requirements Engineering. PhD thesis, TU-Ilmenau (2004)
9. Gosun, A.: Portal of the Video Disc Recorder Project, available online at http://www.vdr-portal.de. Website (March 2010)
10. Riebisch, M., Streitferdt, D., Pashov, I.: Modeling Variability for Object-Oriented Product Lines. In Buschmann, F., Buchmann, A.P., Cilia, M., eds.: ECOOP Workshops. Volume 3013 of Lecture Notes in Computer Science., Springer (2004) 165–178
11. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA, USA (November 1990)
12. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Program. **8**(3) (1987) 231–274
13. OMG: OMG Unified Modeling Language Specification Version 2.2, available online at http://www.omg.org/technology/documents/formal/uml.htm. Website (2009)
14. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, Boston (2004)
15. Olga Fengler, Wolfgang Fengler, V.D.: Transformation zwischen zustandsorientierten Beschreibungsmitteln der Elektronik und Informatik. Deutsche Forschungsgemeinschaft: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen. WILEY-VCH, Weinheim (2002) S. 229–245
16. Jensen, K.: A Method to Compare the Descriptive Power of Different Types of Petri Nets. In Dembinski, P., ed.: MFCS. Volume 88 of Lecture Notes in Computer Science., Springer (1980) 348–361
17. Fengler, W.: A Colored Petri Net Interpretation for Modeling and Controling Textile Processing. CSCW&Petri Net Workshop, 14th International Conference Application and Theory of Petri Nets, Chicago, USA. (1993)
18. IEEE: IEEE standard computer dictionary : a compilation of IEEE standard computer glossaries. IEEE Comp. Soc. Press, New York, NY, USA (January 1991)
19. Poore, J.H.: Introduction to the Special Issue on: Model-Based Statistical Testing of Software Intensive Systems. Information & Software Technology **42**(12) (2000) 797–799
20. Whittaker, J., Thomason, M.: A Markov Chain Model for Statistical Software Testing. IEEE Transactions on Software Engineering **20** (1994) 812–824
21. Prowell, S.J.: JUMBL: A Tool for Model-Based Statistical Testing. Hawaii International Conference on System Sciences **9** (2003) 337c
22. D-MINT: Deployment of Model-Based Technologies to Industrial Testing, available online at http://www.d-mint.org. Website (March 2010)

Model Based Statistical Testing and Concurrent Streams of Use

Frank Böhr *

TU Kaiserslautern
boehr@informatik.uni-kl.de

Abstract. **M**odel **B**ased **S**tatistical **T**esting (MBST) is a highly automated test approach. It allows the fully automated test case generation, execution, and evaluation after building the test model. However, it is not easy to build the model used in MBST if the system under test needs to handle concurrent streams of use, which is the usual case for embedded systems. The usual way to address such a situation is to use strong abstraction, even though it is not impossible to represent concurrent streams of use in the test model used in MBST. The reason to use strong abstraction, is the emerging high complexity of the test model (which arises because of the lack of explicit support of concurrency) and thus its error prone and time consuming construction. This is why this paper focuses on the introduction of an explicit representation of concurrency within the test model. This is done with the use of Petri nets. The paper proposes to use Petri nets as a test model because they are well suited for modeling concurrency and in a second step to generate the test models usually used in MBST based on these Petri nets in order to preserve the ability of statistical analysis.

1 Introduction to MBST

Model **B**ased **S**tatistical **T**esting (MBST) has been used for testing a wide range of applications. These applications vary from sophisticated software engineering environments [1] to data bases [2] and large industrial software systems [3]. MBST is furthermore used in industries and by government agencies [4]. MBST was also used in projects involved with testing embedded systems such as mass storage devices, weapon systems, medical devices and automotive components [5]. More examples can be found in [6][7][8] and [9]. Testing the special characteristics of embedded systems is only mentioned in [10] and [11] but there, the focus is on time and not on concurrency.

Embedded systems are usually handling concurrent streams of use. As an example of two concurrent streams of use imagine a co-drivers power window in a car. It can be controlled by the co-driver himself, but it can also be controlled by the driver. The driver (representing one stream of use) can act concurrently

* I want to thank The Klaus Tschira Foundation gGmbH and Prof. Dr.-Ing. Dr. h.c. Andreas Reuter for funding and supporting my work.

to the co-driver (representing a second stream of use). However, a stream of use might also be representing inputs from a sensor or another software system.

Additionally to handling concurrent streams of use, embedded systems are becoming more and more complex [12] and increasingly software intensive. Even though additional software adds new features to those systems, it introduces potential failures at the same time. An adequate testing approach must consider the special properties of these embedded systems software (e.g. concurrent streams of use).

Poore [4] mentions that the software testing problem is complex because of the astronomical number of scenarios and states of use and that the domain of testing is large and complex beyond human intuition and that statistical principles must be used to guide the testing strategy in order to get the best information for the resources invested in testing because the software testing problem is so complex. MBST offers a solution, because a main benefit of MBST is that it allows the use of statistical inference techniques for computing probabilistic aspects of the testing process, such as reliability [13]. In MBST all possible uses of the software are represented by a statistical model wherein each possible use of the software has an associated probability of occurrence. Test cases are drawn from the sample population of possible uses according to a sample distribution and run against the software under test. The model used to represent the use of the software is a finite state, time homogeneous, discrete parameter, irreducible **M**arkov **C**hain (MC) [9]. This model can be represented as a state transition graph as shown in [13]. The states in this graphical representation determine externally visible states of use. The arcs represent inputs (also called stimuli) to the **S**ystem **U**nder **T**est (SUT) and the required SUT response [7]. This means each arc is tagged with a certain stimulus and with a system response which represent possible uses of the system and its reaction. A user might be a human, a hardware device, another software system, or some combination [4].

An example of a graphical representation of a usage model, which represents the use of a telephone, is shown in Fig. 1. Each usage model has two special states. The first state is called source (On Hook) and the second is called sink (Exit). Each possible walk through the usage model starting at the source state and ending in the sink state is a possible test case [3]. When traversing an arc, the associated stimulus and response are collected. A test case is created by concatenating the stimulus/response pairs of the arcs visited.

Which arc is chosen in a particular state depends only on the sample distribution (shown in brackets in Fig. 1). So each arc is, additional to the stimulus/response pair, associated with a certain probability. These probabilities are used to generate test cases which are representative with respect to the way in which the software will be used after its deployment. The values of the probabilities of outgoing arcs of each state add up to 1 and depend on the current state only. This is normally a reasonable description of the usage of most software systems [15]. Such a model is called a *usage model* because it describes the use of the software rather than the software itself. There are several ways to build the structure of a usage model i.e. a graph like the one shown in Fig. 1

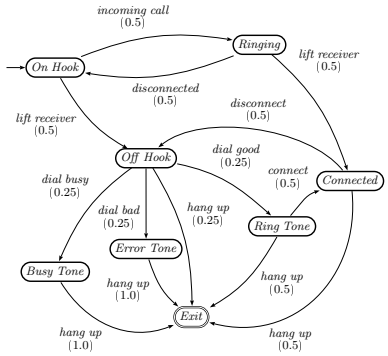


Fig. 1. An example of a usage model [14] which represents the use of a telephone.

without the probabilities. One way is to apply a method called **S**equence **B**ased **S**pecification (SBS). The SBS builds the model structure solely based on the requirements. Approaches introducing hierarchies into usage models are presented in [4][15][16].

A usage model can be represented as a MC. The Markov property of the usage model can be used to compute information which provides a great deal of information about the testing effort [16] like the expected test case length, probability of occurrences for states, long run occupancies of states, arcs and inputs [17], source entropy, amount of statistical typical paths and some more [9] **prior** to any test run and single-use reliability and single-event reliability [17] **after** the test cases are executed.

According to [2] it is possible to obtain the needed probabilities in three ways. The first is called the **un-informed approach**. It consists of assigning a uniform probability distribution across the exit arcs for each state. The second one, called the **informed approach**, can produce many models, and is used when some real usage patterns are available. These patterns could be captured inputs from a prototype or from a prior version of the software.The third approach, called the **intended approach**, is similar to the informed approach in that it can lead to many models, but the sequences are obtained by hypothesizing runs of the software by a careful and reasonable user. A way which allows some degree of automation in the generation of probabilities is presented in [18].

Concurrent streams of use with respect to MBST are only addressed in [19] as far as known to the author. However, the work presented there does not introduce concurrency to the test model. The approach generates test cases (from a usage model like the one in Fig. 1) which each represent a single stream of use, and interleaves these test cases after their generation.

2 Introduction to Petri Nets [1]

Petri nets (PNs) were introduced by C.A. Petri in 1962 [21]. Petri nets are a graphical tool for the formal description of systems whose dynamics are characterized by concurrency, synchronization, mutual exclusion and conflict [22].

The basic graphical representation of a PN is a bipartite directed graph [23] (i.e. it is not allowed to connect places with places or transitions with transitions). It consists of places, transitions, arcs and tokens [24]. Places are drawn as circles and transitions are drawn as bars (or squares) [25]. Arcs are drawn as arrows (with possibly different arrow heads) and tokens are drawn as black circles inside places [26] (or the amount of tokens is written inside the places as a number). An arc can connect a place with a transition or a transition with a place.

Places are used to describe possible local system states. Transitions are used to describe events that may modify the system state. Arcs specify the relation between local states and events [22].

An arc drawn from a place to a transition is called an input arc into the transition from the place. Conversely, an arc drawn from a transition to a place is called an output arc from the transition to the place [23]. Each place connected to a transition t via an input arc is called an *input place* of t, each place connected to a transition t via an output arc is called an *output place* of t and each place connected to a transition t via an *inhibitor arc* (inhibitor arcs are explained at the end of this section) is called an *inhibitor place* of t. A transition is said to be enabled, if all of its input places contain at least one token and all of its inhibitor places contain no token [25]. The behavior of a PN is controlled by the *firing rule*. An enabled transition may *fire* by removing a token from each of its *input places* and depositing a token in each of its *output places* [26]. The tokens which get removed from the Petri net by firing a transition t are denoted by $\mathcal{I}(t)$ (called the input bag of t) and the tokens which get added to the Petri net by firing transition t are denoted by $\mathcal{O}(t)$ (called the output bag of t).

When arc cardinalities (shown as numbers labeling the arcs in Fig. 2) are used, the number of tokens required in each input and inhibitor place for enabling the transition and the number of tokens generated in each output place by the transition firing are determined by the cardinality of the arc connecting the place and the transition [22]. The number of tokens in each place is called the *marking* of the PN and is denoted by μ. It is possible to think of the marking as a vector of non negative integers. Each element in the vector belongs to the amount of tokens in one particular place. Thus adding (subtracting) the output bag (input bag) of a transition to (from) a marking can be done by adding (subtracting) vectors (see line 8 of algorithm 1).

A special type of arc is the *inhibitor arc*, which is represented as a circle-headed arc originating from a place and ending at a transition [22]. If an inhibitor arc exists from a place to a transition, then the transition will not be enabled

[1] Fig. 2 in this section and Fig. 3 and Fig. 4 in the next section are created using a tool called "yEd Graph Editor" available from [20].

if the corresponding inhibitor place contains more or equal tokens than the arc cardinality [27]. The use of inhibitor arcs allows implementing a test for zero tokens in a place which is not possible with normal arcs in general. The addition of inhibitor arcs is an important extension of the modeling power of PNs, which gives them the same modeling power as Turing machines [22]. With the above extended Petri nets powerful enough to simulate Turing machines, all nontrivial problems for such Petri nets become undecidable [28].

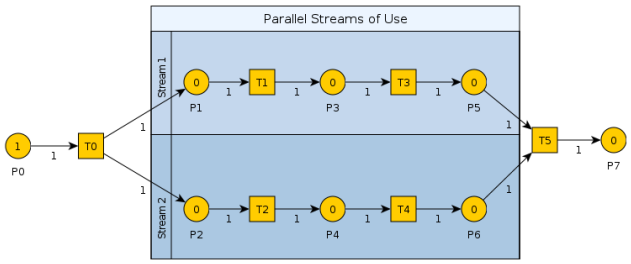


Fig. 2. An example Petri net which shows two streams of use.

An example Petri net is shown in Fig. 2. It consists of eight places numbered $P0$ to $P7$. Each place contains 0 tokens except $P0$ which contains 1 token. The Petri net consists furthermore of six transitions numbered $T0$ to $T5$ and 14 arcs (arrows) each having a cardinality of 1. The box in the background does not belong to the Petri net and is only used to highlight the concurrent parts of the Petri net.

3 Petri Nets and Usage Models

A usage model can be seen as a special kind of Petri net. It is a Petri net which has only one token in each of its possible markings. This token indicates the current state of use in the usage model. The relation between Petri nets and usage models gets more obvious if the stimulus/response pairs which are associated to the usage model arcs get represented as Petri net transitions (see Fig. 3). The restriction to use only one token is not needed during modeling i.e. creation of the test model. However, it is needed during the statistical analysis of the test results, because it allows to represent the usage model as a markov chain.

This is why this section explains first how it is possible to use the modeling capabilities of Petri nets during the usage model building and, second a way how it is possible to transform the created model to a Petri net which uses only one token in order to preserve the ability for statistical analysis of the test results.

The ability to model concurrent behavior is an integral part of Petri nets. As an example see Fig. 2. Place $P0$ contains 1 token. It is the only input place

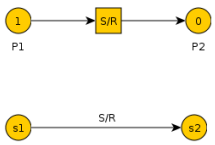


Fig. 3. The lower part shows a tiny usage model with two states. Changing the state from $s1$ to $s2$ generates a test case with stimulus S and response R. The upper part shows a Petri net which does exactly the same.

of transition $T0$ and the arc connecting $P0$ and $T0$ has cardinality 1. Thus transition $T0$ is enabled and can fire. Firing transition $T0$ results in removing the token from $P0$ and putting 1 token in each of the output places of $T0$ (i.e. $P1$ and $P2$) because each arc connecting $T0$ to an output place has cardinality 1. This situation starts two concurrent streams of use. These streams are $T1$, $T3$ and $T2$, $T4$. The involved transitions can happen in any of the following orders: ($T1$,$T3$,$T2$,$T4$),($T1$,$T2$,$T3$,$T4$),($T1$,$T2$,$T4$,$T3$),($T2$,$T1$,$T3$,$T4$), ($T2$,$T1$,$T4$,$T3$),($T2$,$T4$,$T1$,$T3$) which is basically each possible interleaving of the two streams of use.

As an example consider ($T1$,$T2$,$T4$,$T3$). Firing of $T1$ removes one token from $P1$ and adds one token to $P3$ which results in enabling $T3$. The token in $P2$ which remains there from firing $T0$ enabled $T2$ whose firing removes the token from $P2$ and puts one token in $P4$ which enables $T4$. Firing $T4$ moves the token from $P4$ to $P6$. In this situation firing $T5$ is not possible because there is no token in $P5$. Thus the only remaining enabled transition $T3$ is firing and moves a token from $P3$ to $P5$. In this situations there is one token in $P5$ and one token in $P6$ which enables $T5$. Firing $T5$ is the end of the concurrent streams of use.

The association of stimulus/response pairs to Petri net transitions allows the creation of a test model with concurrent streams of use.

Modeling each of the possible interleavings is possible with a usage model as well, but the resulting model is much more complicated even in the case of the small example shown in Fig. 2. The usage model is shown in Fig. 4. The concurrency cannot be easily seen in the usage model. Additionally, changing one of the streams of use (e.g. adding one more transition) results in additional usage model states and additional usage model arcs originating from various different usage model states and cannot be performed easily.

Even though the usage model is more complicated it is well suited for generating test cases and especially for performing statistical analysis. This is why this paper proposes to use Petri nets during modeling and usage models during test case generation and statistical analysis. To achieve this goal it is necessary to transform Petri nets to usage models after building the test model (i.e. Petri net) and before test case generation and statistical analysis. How this is possible gets explained in the following part of this section and is shown as pseudo code in algorithm 1.

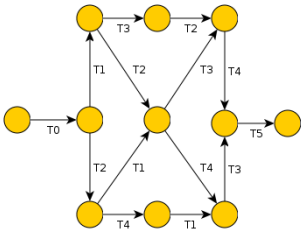


Fig. 4. A usage model which represents the same possible uses as the PN in Fig. 2.

Basically, the idea is to generate the reachability graph of the Petri net. One state in the reachability graph corresponds to one possible marking of the Petri net. Each arc in the reachability graph is tagged with the transition which caused the state change. The resulting reachability graph is the usage model. The probability to choose a certain usage model arc can be computed based on transition weights. This means that each transition in the Petri net gets associated with a certain weight, which is a non negative integer value. The probability to chose and fire a transition among possibly many other enabled transitions (which is the probability to chose a certain arc in a certain usage model state) gets computed by dividing the weight of the chosen transition by the sum of the weights of all enabled transitions.

Algorithm 1 works as follows. The outputs are the two set $\mathcal{S}$ and $\mathcal{P}_{\mathcal{S}}$. $\mathcal{S}$ contains the states of the usage model and $\mathcal{P}_{\mathcal{S}}$ contains the state transitions between these usage model states. These sets are initially empty as indicated in the second line of the algorithm.

The set $\mathcal{S}^{\text{next}}$ contains the known but unprocessed usage model states. One usage model state belongs to one Petri net marking. Initially there is only one known state and this state is unprocessed as shown in the third line. This state corresponds to the initial marking of the Petri net.

The while loop in line 4 goes on until there are no more states in $\mathcal{S}^{\text{next}}$ i.e. until all possible markings (i.e. usage model states) got processed.

The first step in the while loop is to chose one known unprocessed state s_i from $\mathcal{S}^{\text{next}}$ which is going to be processed next (line 5).

This state gets removed from $\mathcal{S}^{\text{next}}$ in line 6 in order to avoid processing it again in a later iteration of the while loop.

The for loop in line 7 chooses one transition from the set $\mathcal{E}(s_i)$ which represents all enabled transitions (i.e. transitions which can fire) in state s_i.

The first line in the for loop fires the chosen transition by subtracting the input bag and adding the output bag of the chosen transition to the marking associated to s_i (i.e. μ_i). The resulting marking μ_{i+1} corresponds to a new state s_{i+1} which can be reached from s_i by firing transition t^j.

The probability to fire t^j in state s_i gets computed in line 9 of the algorithm. It divides the weight of transition t^j by the sum of the weights of all transitions in $\mathcal{E}(s_i)$.

Lines 10 to 13 are skipped if the reached state s_{i+1} was already reached in a previous iteration of the while loop. s_{i+1} gets added to the set of known states $\mathcal{S}$ and the set of known but unprocessed states $\mathcal{S}^{\text{next}}$ if the state wasn't reached up to now.

Line 14 adds the found state transition to the set $\mathcal{P}_{\mathcal{S}}$. A state transition consists of a start state (s_i), an end state (s_{i+1}), the associated transition (t^j) which represents the associated stimulus/response pair and the probability to choose the state transition (f^j).

The next step in line 15 is to go back to the start of the for loop in line 7 and to choose the next transition which is fired in marking μ_i which belongs to s_i. The for loop goes on firing transitions in this state until each transition in $\mathcal{E}(s_i)$ gets fired. Thus each state reachable from s_i gets reached.

The while loop chooses the next state which is going to be processed from $\mathcal{S}^{\text{next}}$ after the for loop fired each possible transition in state s_i.

The algorithm goes on like this until there are no more known and unprocessed states. In that case the algorithm terminates and reaches line 17. In this case $\mathcal{S}$ contains each reachable state and $\mathcal{P}_{\mathcal{S}}$ contains all possible state transitions. The algorithm does not terminate if the state space of the Petri net is infinite.

Algorithm 1 This algorithm generates a usage model out of a Petri net. Some parts of the algorithm and of its notations are adapted from [29].

1: **procedure** generateUsageModel(**out**: $\mathcal{S}, \mathcal{P}_{\mathcal{S}}$)
2: $\mathcal{S} = \varnothing$; $\mathcal{P}_{\mathcal{S}} = \varnothing$;
3: $\mathcal{S}^{\text{next}} = \{s_0 = (\mu_0)\}$;
4: **while** $\mathcal{S}^{\text{next}} \neq \varnothing$ **do**
5: choose a state $s_i = (\mu_i)$ from $\mathcal{S}^{\text{next}}$
6: $\mathcal{S}^{\text{next}} = \mathcal{S}^{\text{next}} \backslash \{s_i\}$;
7: **for all** $t^j \in \mathcal{E}(s_i)$ **do**
8: $s_{i+1} = \mu_{i+1} = \mu_i - \mathcal{I}(t^j) + \mathcal{O}(t^j)$;
9: $f^j = \hat{w}(t^j, \mathcal{E}(s_i))$;
10: **if** $s_{i+1} \notin \mathcal{S}$ **then**
11: $\mathcal{S} = \mathcal{S} \cup \{s_{i+1}\}$;
12: $\mathcal{S}^{\text{next}} = \mathcal{S}^{\text{next}} \cup \{s_{i+1}\}$;
13: **end if**
14: $\mathcal{P}_{\mathcal{S}} = \mathcal{P}_{\mathcal{S}} \cup \{(s_i, f^j, t^j, s_{i+1})\}$;
15: **end for**
16: **end while**
17: **end procedure**

4 Conclusion

MBST has a high degree of automation. It allows achieving fully automated test case generation, test execution, test evaluation and reliability estimation

based on a usage model [6]. However, it was hardly possible to consider concurrent streams of use in MBST without using strong abstraction up to the presented proof of concept, because of the emerging complexity of the usage model. Handling concurrent streams of use is important in embedded systems testing because it is usual that these systems need to handle more than one stream of use. The ability to model several streams of use is achieved by the use of Petri nets. It is important to notice that a usage model can be automatically generated out of a Petri net. Thus all advantages of MBST are preserved e.g. generating test cases which include the test oracle (this means it is possible to decide between pass and fail of a generated test case automatically) and the ability to statistically analyze the results and estimate the reliability of the SUT.

The presented work furthermore allows preserving the efforts invested in previous usage model construction. This is the case because a usage model can easily be turned into a Petri net as shown in Fig. 3. Concurrent streams of use can be added to the usage model after converting it to a Petri net.

The next step is evaluating the method at an example of realistic size like the one presented in [6]. Tool support for generating a usage model out of a Petri net and for generating test cases out of a usage model and statistical analysis is available.

References

1. Gwendolyn H. Walton, J. H. Poore, and Carmen J. Trammel. Statistical testing of software based on a usage model. *Softw. Pract. Exper.*, 25(1):97–108, 1995.
2. J. H. Poore and James A. Whittaker. Markov analysis of software specifications. *ACM Trans. Softw. Eng. Methodol.*, 2(1):93–106, 1993.
3. T. Bauer, H. Stallbaum, A. Metzger, and Robert Eschbach. Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest. In *GI-Edition - Lecture Notes in Informatics (LNI) - Proceedings 121*, pages 99 – 111, München, February 2008.
4. J. Poore and C. Trammell. Application of statistical science to testing and evaluating software intensive systems. In *Science and Engineering for Software Development: A Recognition of Harlan D. Mills Legacy.*
5. Jason M. Carter and Jesse H. Poore. Sequence-based specification of feedback control systems in Simulink. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 332–345, Richmond Hill, Ontario, Canada, 2007. ACM.
6. Thomas Bauer, Frank Böhr, Dennis Landmann, Taras Beletski, Robert Eschbach, and Jesse Poore. From requirements to statistical testing of embedded systems. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, page 174. IEEE Computer Society, 2007.
7. Thomas Bauer, Frank Böhr, and Robert Eschbach. On MiL, HiL, statistical testing, reuse, and efforts. In *1st Workshop on Model-based Testing in Practice - MoTiP*, Berlin, June 2008.
8. K. Agrawal and J.A. Whittaker. Experiences in applying statistical testing to a real-time, embedded software system. In *Proceedings of the Pacific Northwest Software Quality Conference*, pages 154–170, Portland, 1993.

9. Kirk D. Sayre. *Improved Techniques for software testing based on markov chain usage models*. PhD thesis, The University of Tennessee, 1999.
10. Frank Böhr. Model based statistical testing and time. In *2nd Workshop on Model-based Testing in Practice - MoTiP*, pages 33–42, Enschede, June 2009.
11. Frank Böhr. Model based statistical testing and durations. In *17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 344 – 351, Oxford, March 2010.
12. Frank Böhr. *Verhaltensbasierte Ansätze in verlässlichen Automotivesystemen: Anwendung und Evaluierung*. Vdm Verlag Dr. Müller, February 2008.
13. James A. Whittaker and Michael G. Thomason. A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.
14. S.J. Prowell. Jumbl: a tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 9 pp., 2003.
15. J. Runeson and C. Wohlin. *Usage Modelling: The Basis for Statistical Quality Control*. Annual Software Reliability Symposium. Denver, Colorado, USA, 1992.
16. Richard C. Linger and Carmen J. Trammel. Technical report cmu/sei-96-tr-022 esc-tr-96-002. Tchnical teport, CMU/SEI, November 1996.
17. S. J. Prowell. Computations for markov chain usage models, 2000.
18. Ibrahim El-Far. *Automated construction of software behavior models*. PhD thesis, Florida Institute of Technology, 1999.
19. S. J. Prowell. Using markov chain usage models to test complex systems. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, volume 9. IEEE Computer Society, 2005.
20. http://www.yworks.com/en/index.html.
21. Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
22. D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and Giuseppe Conte. *Modelling with Generalized Stochastic Petri Nets*. Turino, 1 edition, February 1994.
23. Muppala Jogesh K., Kishor S Trivedi, and Gianfranco Ciardo. Stochastic reward nets for reliability prediction. In *Communications in Reliability, Maintainability and Serviceability*, volume 1 of *2*, pages 9 – 20, 1994.
24. Claude Girault and Valk Rüdiger. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer, 1 edition, December 2002.
25. C. Lindemann, G. Ciardo, R. German, and G. Hommel. Performability modeling of an automated manufacturing system with deterministic and stochastic petri nets. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 576–581 vol.3, 1993.
26. C. Ciardo, A. Blackmore, P.F.J. Chimento, J.K. Muppala, and K.S. Trivedi. Automated generation and analysis of makov reward models using stochastic reward nets. *Linear Algebra, Markov Chains, and Queueing Models*, 48:145–191, 1993.
27. Robert Zijal and Reinhard German. A new approach to discrete time stochastic petri nets. In *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, volume 199/1994 of *Lecture Notes in Control and Information Sciences*, pages 198–204. 1994.
28. Hsu-Chun Yen. Introduction to petri net theory. In *Recent Advances in Formal Languages and Applications*, volume 25/2006 of *Studies in Computational Intelligence*, pages 343–373. Springer Berlin / Heidelberg, 2006.
29. Robert Zijal and Gianfranco Ciardo. DISCRETE DETERMINISTIC AND STOCHASTIC PETRI NETS. Technical report, Institute for Computer Applications in Science and Engineering (ICASE), 1996.

A Trial on Model Based Test Case Extraction and Test Data Generation

Xiaojing Zhang and Takashi Hoshino

NTT Cyber Space Laboratories,
1–1 Hikarinooka, Yokosuka-Shi, Kanagawa, 239–0847 Japan
{zhang.xiaojing,hoshino.takashi}@lab.ntt.co.jp
http://www.ntt.co.jp/cclab/e/

Abstract. For software quality assurance, it is necessary to improve the testing process, which ensures that software works correctly. As the task of test design, such as creating the test cases and the test data, are time and effort consuming when done manually, we propose a technique that can generate these artifacts from UML design models, aiming to improve SUT coverage and test density within limited man-hour. We implemented a prototype of a test design support tool, carried out a case study for evaluation through the comparison between automatic generation and manual derivation.

Key words: model based testing, test case extraction, test data generation, UML 2.0, activity diagram

1 Introduction

1.1 Background

Software defects, which lead to economic loss, have become an unignorable object of public concern. In this study, we aim to achieve better quality assurance by improving testing, within a reasonable range of the quality-cost-delivery balance. We consider testing as a confirmation of whether the product is developed just as one intended, rather than just a search of bugs. Here, the intentions of the customer are formalized into software design. So the subject of our study are tests which confirm "whether the product is implemented as its design".

To achieve test improvement, it is essential to increase the quantity and quality of the test, besides other related factors such as testing process and testing management. We use test density as a metric of test quantity, which is represented by the following formula where SUT represents software under test:

Test density = Number of test cases ÷ Size of the SUT.

On the other hand, SUT coverage can be used as a metric of test quality. Here, we consider 2 kinds of SUT coverage, structure coverage and input space coverage.

Structure coverage is percentage of SUT's elements which have been tested. This can be represented as below:

Structure coverage = Elements tested ÷ Total number of elements.

Here, the elements that make up the structure of the software are instructions,

branches, paths, methods, classes, components, etc. In this study, we focus on the path coverage.

Input space coverage, is the percentage of inputs given to the SUT which are used for testing. This can be represented as below:

Input Space Coverage = Inputs used for testing ÷ Entire input space.

Here, the entire input space is a collection of all the possible inputs that can be given to the SUT. Choosing inputs is important for quality assurance, because the number of inputs is usually infinite but we can only do testing finite times.

When improving test density and SUT coverage, we also need to work on test laborsaving. Pursuit of high density and coverage will ask for more manpower of testing. So in order to achieve these goals within constraints on costs and delivery time of development, "easier way" of testing is strongly required.

Therefore, as a part of test laborsaving, our study focuses on test design automation. Test design is composed of activities extracting test cases and test data from the software design. The test data materializes a test case and makes it executable. Unlike test execution, test design can only be done by a few skilled engineers, so it may become the bottle neck of the test process. In this study, our goal is to increase the test density and SUT coverage while reducing man-hours of test, by means of test design automation.

1.2 Motivation

As an approach of test design automation, we proposed a method to automatically generate test cases and test data as test design artifacts, based on the software design described in UML (UML model). In particular, the inputs of this generation are UML 2.0 class diagrams and activity diagrams, and outputs are test cases, and test data with hierarchical structure such as object data-type. We give 3 motivations below to explain why we choose this approach.

Easy notation. Existing research on test design automation especially test case extraction, as mentioned in literature[7, 4, 2], mainly aims at embedded systems and communications protocols which are based on state transition model, so the applicable scope is still limited. Besides, remember that the input of test design is software design, means formalized users and developers' intention. But there is also a problem that software design notation proposed is often "original" and "esoteric", therefore making a high barrier for the technique's practitioners.

In consideration of usability, our study adopts UML which has been widespread in recent years as a notation for software design, especially activity diagrams and class diagrams which have been widely experienced by the developers. UML 2.0 is well suited for the formalization of software design information and it also has a high affinity with mechanical processing.

Software design based. As a part of test design, to consider the variation of test data and to create the data instances themselves, will be very labor-consuming if you want to increase SUT coverage and test density. Because, you need to create a huge amount of test data, and before that you have to consider about selecting the input values, which should cover the input space comprehensively and efficiently.

In contrast, existing research on test data generation, as surveyed in literature[5, 3, 6], has the following problems. A lot of studies generate test data from source code, but studies using software design as input are still few. Also, there are some tools that generate pseudo personal information such as combination of name and address, etc.[1] These kind of data are useful but they are randomly generated from dictionary, not based on design information. Unlike software design, source code and dictionaries might be away from the user/developer's intention, so the validity of the content has to be verified.

Structured test data. In addition, about the data-type which can be generated, studies dealing with only primitive type for example integer and boolean are the majority. As for managing the variation of the data-type, it is still developing.

We consider that in the test data generation, how to deal with test data that has complicated structure is the problem. Therefore this study focuses on test data generation techniques dealing with the data-type with hierarchical structure (e.g., object data-type).

Data-types such as XML and objects are often used in recent software, so it is insufficient to only have primitive type of the numerical value and the character string as the test data. we think by coping with structural data-type, it may increase the applicability of generation techniques for more practicable tests. Especially, new ideas will be necessary because more variations have to be considered when the test data has hierarchical or repetition structure.

This paper makes the following contributions: First, we present a technique that achieves higher test density and SUT coverage by automatically generating test cases and test data from UML design models. It can handle data-types with hierarchical structure, and it can generate abnormal test data therefore it's more useful in real project. Second, we evaluated our approaches with one Java web system, and get some useful lessons including the correspondence and differences between manual creation and automatic generation.

The remainder of this paper is organized as follows. Section 2 describes test case and test data generation techniques. Section 3 evaluates our method by a case study. Section 4 shows the future works and concludes the study.

2 Generation techniques

2.1 Test case extraction

IEEE Standard 610 defines test case as follows:

> A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

In this study, we consider a test case to be a pair of (a) a particular execution path within the intended behavior, and (b) the test data including the input and the expected result.

(a) Execution path
First, we analyze the structure of the input UML design model and decompose it smaller and smaller into components, classes, methods and activities. Then we extract the execution paths from each activity, which is a directed graph, by simple depth first search algorithm. This is a search of activity final node starting from initial node.

In the extraction of the execution paths, all possible execution paths are extracted exhaustively, to make the structure coverage secured.

However, when a loop exists in the activity, infinite execution paths can be extracted. We limit it to two paths in case of not passing the loop and the case of passing the loop just once. There for the number of paths extracted can be represented as a sum of permutations shown below, when N is the number of loops within the activity:

$$\text{Number of paths} = \sum_{i=0}^{N} {}_N\mathrm{P}_i$$

(b) Test data
The other element of the pair is the test data, being necessary and indispensable to extract executable test cases. therefore for each extracted execution path, we have to generate the test data that can trace this path, and we will describe the generation procedure in the next section.

2.2 Test data generation

The test data generation method is the one according to the idea of SUT coverage especially input space coverage. In order to generate the test data with high coverage, we need to obtain thoroughly the factors and the levels that constitutes the test data. In software testing, the factors refer to the variable to be considered, while the levels mean the values set to a specific variable. For instance, we can think about two levels (male and female) for the factor of sex.

The test data is generated with the 4 steps explained one by one below, and an UML model(class diagram, activity diagram only) is assumed to be the input of this.

(1) Extraction of the factors we acquire the activity parameter nodes which are the input parameters of the activity, and these are assumed to be factors. The input parameters have a name and a type.

(2) Generation of the test data specification The test data specifications are a set of constraints that the test data should follow in order to trace one execution path of SUT. A test data specification is generated per each factor of each execution path. In other words, it contains the information of "what variable x should be proper for path y?"

(a) Data structure

The test data specification maintains the domain of the factor internally. Figure 1 shows the structure of the test data specification.

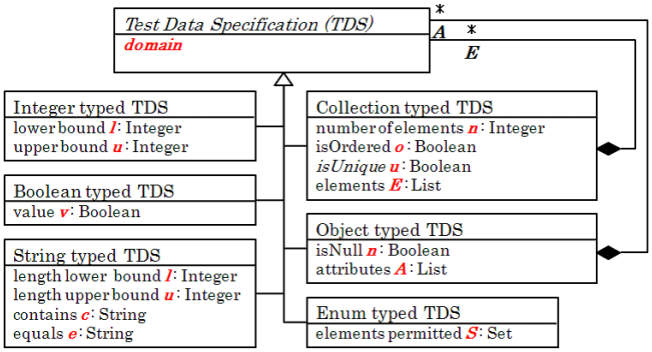


Fig. 1. Data structure of the test data specification

We enable the generation of the test data with the hierarchical structure, by defining the test data specification corresponding to the factor of object data-type. As shown in Figure 1, the "object typed test data specification" contains a list $\boldsymbol{A}$ in its domain, and the list consists of the test data specifications of its own attribute. As a result, we can recursively specify the constraints which the object data-type factors should fill, therefore we can handle the hierarchical structure. Moreover, for collection type, we consider that it is possible to handle the repetition structure by preserving the test data specification for each element in the collection.

The domain's form of each test data specification is different and depends on the data-type of the factor. For instance, the factor of the Integer data-type is associated with the Integer typed test data specification, and this specification has a domain defined by two parameters like lower bound $\boldsymbol{l}$ and upper bound $\boldsymbol{u}$. In addition, each kind of domain has an invariant condition. An invariant condition is a constraint that the domain should always satisfy. For instance, "$\boldsymbol{l} \leq \boldsymbol{u}$" should always be true in integer typed test data specification.

(b) Extraction

The extraction of the test data specification contains 2 steps, the initialization and the update.

The initialization is explained as follows. For one specified factor in a certain execution path, we acquire the "type" of the factor according to the UML meta model. And then, a test data specification corresponding to the type of the factor is generated. For example, the string typed test data specification is generated for a String data-type factor. Next, an initial value is set to the domain of the generated test data specification. For instance, lower bound $\boldsymbol{l}$ and upper bound $\boldsymbol{u}$ included in the Integer typed test data specification will be set to MIN and MAX value of the system default. However, if the data definition concerning the type of the factor is described as an invariant condition in the UML model (constraints associated to the class in the class diagram), we give higher priority to this data definition and use it as the initial value of the domain.

Next, the update is explained as follows. In this step, we clarify the conditions in order to trace down the execution path, and update the test data specification.

The execution path is traced from the initial node to the activity final node, and the preconditions of activity, the local preconditions and the guard conditions in the execution path, are processed sequentially. For instance, assume that when scanning through a certain execution path, we met a guard condition says "name.length $<$ 20". This condition refers to the factor "name", so we update (overwrite) length upper bound $\boldsymbol{u}$ to 19, in the domain of the test data specification of "name". Such an update is repeated until all conditions are processed and the activity final node is reached.

(3) Generation of the levels Several (finite) concrete values selected from the domain, which will be set to a factor, are assumed to be the levels. In the extraction of the levels, we basically apply the technique of equivalent classes division and the boundary value analysis. In consideration of the input space coverage, we extract not only the values which are inside of the domain as normal levels but also the values which are outside of the domain as abnormal levels.

As for object data-type, if every attribute is "normal", then the whole object is assumed to be a normal level. Otherwise, if any of the attributes is "abnormal", then the whole object is assumed to be an abnormal level. In a word, after expanding an object to the tree structure hierarchically, if as much as one "abnormal level" exists as a "leaf", then the entire object is assumed to be abnormal.

(4) Combination of the levels The test data is composed of the input and the expected result.

In general, the input is a combination of the levels set to factors respectively. Since testing all combinations is unrealistic, there are a variety of techniques to reduce the combinations. Here, we adopt a simple goal, "level coverage", to generate inputs making all levels generated be covered. This means, each level only has to be used at least one time somewhere in the test data. In our method, the combinations that consist all of normal level are generated, and these are assumed to be normal inputs. Because we generate normal inputs in minimum number that can satisfy level coverage, the number of normal inputs is same to the number of normal levels, of the factor which has the most normal levels. On the other hand, the combinations where only one abnormal level included are also generated, and these are assumed to be abnormal input. These tests aim to confirm that an abnormal input CANNOT trace the execution path. Only one abnormal level is included to make the fault localization easier when failure occurs.

The expected result corresponding to an input, is acquired from the postcondition associated to the execution path.

3 Evaluation

3.1 Tool prototype

To verify and evaluate the technique proposed above, we implemented a prototype of "test design support tool".

The software design model based on UML 2.0 in XMI 2.1 format (a xml file) is assumed to be the tool's input, and the output are the list of test cases and the test data used by each test case (csv files). A test data consists of an ID, input values, an expected result, and a boolean value which shows whether it's normal for the according path.

3.2 Viewpoints

We evaluated whether the proposed method, aiming to the automation of the test design, is effective enough to secure the test density and the SUT coverage by few man-hours. First we create an UML model of an application for evaluation. and then we compare the test design artifacts made from the same UML model, both created by hand (manual) and by our generation technique (automatic).

The viewpoints of this evaluation are shown below:

Comparison of the manpower consumed for the test case extraction and the test data creation.

Comparison of the test density and the SUT coverage of a part of test cases which have correspondence between "manual" and "automatic". Here, when we can judge that 2 test cases are doing the same test, we call they have correspondence.

3.3 Results

We carried out our evaluation on an Java web application, which is a shopping store. We used the serverside order processing component of the application, and the scale is about 4.8 KLOC.

As the comparison result between manual and automatic, number of test cases are shown in table 1, and the test density are shown in table 2. Here in table 2, "Automatic" is calculated using number of automatic test cases WITH correspondence (360 in table 1).

The structure coverage of test cases and input space coverage of the test data are shown in table 3.

We used one average skilled engineer for this evaluation and the manpower cost totally is shown in table 4. The unit man-power cost per test case and per scale are shown in table 5. Note that our prototype generates test cases within one minute so we did not count it in table 4. The overhead cost of automatic means the cost of retouching existing "rough sketch" UML model more detailed enough to generate test cases, this assuming to be 50% of the overall cost of UML modeling. In the test execution after the test design, cost of test case selection will also be the overhead. Here, we didn't consider it because no evaluation on that part was carried out yet.

As a result of the evaluation, by the proposed technique, for 56.3% of manual test design work, 52% of its manpower can be reduced. And beside this, the test density and the SUT coverage can be highly improved.

Table 1. Comparison of the number of test cases

	Manual	Automatic
test cases WITH correspondence	107 (56.3%)	360 (6.0%)
test cases WITHOUT correspondence	83 (43.7%)	5597 (94.0%)
total	190	5957

Table 2. Comparison of the test density

	Manual	Automatic
test density (number of test cases / KLOC)	39.6	75.0

Table 3. Comparison of the SUT coverage

		Manual	Automatic
structure coverage		covered all execution paths	covered all execution paths
input space coverage	levels	representative values only	covered all boundary values
	combinations	without clear rules	covered all levels

Table 4. Comparison of the manpower

	Manual	Automatic
on test design (man-minutes)	2330	0
on other overheads (man-minutes)	0	1123
total (man-minutes)	2330	1123

Table 5. Comparison of unit man-power

	Manual	Automatic
manpower per test case (man-minute / test case)	12.3	3.1
manpower per scale (man-hour / KLOC)	8.1	3.9

3.4 Discussion

The proposed technique achieved certain effect to improve the test density and the SUT coverage by less manpower. However, some problems have been left, as the technique could not correspond to a part of manual made test cases, and also the number of generated test data is too large to be used in test execution. We will give a more detailed discussion based on the evaluation results as follows. We separate those test cases which have correspondence between automatic and manual, which means the part could be improved by automation, and those without correspondence.

Test cases WITH correspondence
These are the test cases created both by the engineer and our tool. Most tests of the normal inputs to the execution paths, belong to this category. The test density has improved by automations, but we still need to improve the method by introducing orthogonal tables or pair-wise methods to narrow the number of combinations in the future. On the structure coverage, all the execution paths

are covered both by the automatic tool and the manual operation. But about the input space coverage, several differences are observed at the following points:
(1) Extraction of levels
For instance, when a constraint "arbitrary character string" is specified, only one test data is created by the engineer manually, though in automatic generation the test data are generated with two or more variation, changing the length of the character string.
(2) Combination
Because the man-hour is limited at the manual creation, the variations of the test data containing other than "key" factor which influences the branch of the execution paths explicitly, are less considered (only one to several). On the viewpoint of whether it can comprehensively confirm that software works correctly according to the design, we can say that manually created test data are inferior to the automatically generated test data.

Manual test cases WITHOUT correspondence to automatic
43.7% of the test cases created manually could not be generated by the automatic tool. The typical reasons are listed below:
(a) A part of information is not described in the UML model.

Generally, it is difficult to include information related to semantics of the software into the UML model, and so often not to be described. For example, a variable of string type though only numbers will be inputed actually, or various dependencies exist among two or more factors etc. It seems that it is possible to deal with this kind of knowledge by deciding a notation and describing them in the UML model. Or, when necessary information is formalized outside of the UML model, a method is needed to extract test cases considering this information.
(b) Two or more activities are not processed integrated.

Because human can access to information in the UML model comprehensively, they make test cases even considering the internal branch of the subactivity called by the test target activity. Of the current state, our proposed method pays attention only to one activity and does not consider the nested structure of activities. To mimic human behavior, test cases extraction based on the viewpoint of integration test will be requested in the future.

Automatic test cases WITHOUT correspondence to manual
Almost abnormal inputs automatically generated belong to this category. Abnormal (error handling) tests created manually are just like the "null" tests, but lots of abnormal variations are outputted in automatic generation. For instance, to say nothing of "null", all things contains abnormal value at the leaf level somewhere in the tree structure of an object, are outputted as an abnormal test data. However, as well as the test by normal inputs, "many more" does not mean "better". It is necessary to narrow down the number of test cases by an abnormal input, to a practicable level. For example, for object data-type, an improvement can be done by outputting not all leaf level abnormal values, but the important ones only while securing input space coverage.

4 Future works and conclusions

In this paper, we confirmed the effectiveness and the insufficient points of the proposed method by using a "toy application" with a relative small size. We need to conduct an application experiment on a real, alive project, to get more reliable evaluation results.

Moreover, we need to abolish various constraints gradually, to improve the proposed technique. The priority of the constraints will be clarified by the result of the application experiment to a real project.

The main constraints include the following. (1) Cannot treat other variables except the input parameters of an activity, as factors. (2) Cannot handle the conditions describing the dependence between two or more factors. (3) Cannot handle the case when the value of the factors are overwritten and changed during the execution path.

In addition, by test design automation we are able to easily generate a large number of test cases, but if it takes a great deal of manpower to conduct all the tests generated, it is not significant at all. Therefore, we need to discuss the cooperation of test design automation tools and test execution automation tools such as xUnit.

In conclusion, we proposed a method to extract test cases and test data with hierarchical structure, from UML design models. And we evaluated the method by a tool prototype. As a result, in a part of the test design, higher SUT coverage and the test density are achieved by fewer man-hours, compared with manual creating. We wish to contribute more to the software quality assurance through the further test design automation techniques in the future.

References

1. Black Sheep Web Software: generatedata.com, http://www.generatedata.com/
2. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies. pp. 31–36. ACM (2007)
3. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the 2nd Conference on Computer Science and Engineering. pp. 21–28. Linkoping (1999)
4. Hartman, A.: AGEDIS-Model Based Test Generation Tools. Agedis Consortium (2002)
5. Korel, B.: Automated software test data generation. Software Engineering, IEEE Transactions on 16(8), 870–879 (Aug 1990)
6. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification & Reliability 14(2), 105–156 (2004)
7. Prasanna, M., Sivanandam, S., Venkatesan, R., Sundarrajan, R.: A survey on automatic test case generation. Academic Open Internet Journal 15 (2005)

SESSION 2
Academic model-based test tools

Fokus!MBT – A flexible and extensible toolset for Model-based testing approaches

Marc-Florian Wendland

Fraunhofer Institut FOKUS, Modeling and Testing Department,
Kaiserin-Augusta-Alle 31, 10589 Berlin, Germany

Marc-Florian.Wendland@fokus.fraunhofer.de

Abstract. Model-based testing (MBT) is a test engineering approach that facilitates the development and documentation of test processes by including formal models and automatically executed transformations between them. This offers a great potential of test automation, which is one of the most promising benefits of MBT. Similar to the concepts of Model-Driven Engineering (MDE), the idea of MBT includes an abstracted view on the problem domain. Details which are not necessary for the testing process can be omitted in order to get a simplified representation of what should be tested. This abstract view is concretized by subsequently applied transformations or manual augmentation. Fokus!MBT is a set of compound tools supporting the model-based paradigm for testing by providing an adaptable infrastructure. Internally, it is based on a dedicated metamodel for testing (TestingMM) that allows the description of an entire test process within one single artifact.

1 Architecture

Fokus!MBT defines a compound infrastructure of independent and heterogeneous tools that are connected with each other in order to be realize a specific, model-based testing process. It consists mainly of two parts, that is a core component and the TestingMM, fully integrated into the Eclipse IDE. The highly extensible core component is responsible for the integration of futurity extensions as well as the overall process flow among the participating tools. Therefore, it defines a set of extension points that can be used by tool providers to integrate their implementation easily into Fokus!MBT, whereas the core component itself merely offers a set of common functions (like OCL guidance or test data generation). Fokus!MBT is built on top of the ModelBus (http://www.modelbus.de) middleware platform.

The TestingMM is represented as an Ecore-based metamodel. It implements the UML Testing Profile standalone metamodel and reuses several concepts of the UML superstructure as well as of Eclipse's TPTP model. The TestingMM consist of ten subpackages, including concepts for test design, test data, test purposes, test behavior, test execution, etc. that are suitable to describe an entire test process. Within Fokus!MBT the TestingMM acts as canonical data model for the integrated tools.

2 Methodology

Fokus!MBT is highly aligned with UML and the UTP, for which it provides a built-in implementation of the profile. This enables a tester to either augment an existing system model with test relevant information or to build a pure testing model from scratch based on UTP. The benefits of these approaches are that testers on the hand can make use of plenty freely available UML tools. On the other hand, the communication between testers and architects is often improved since both parties use the same language. The conceptual and technical gap between testing and developing engineers is bridged by the use of UML as common notation.

3 Built-in Core Functionality

3.1 Requirements Traceability

Fokus!MBT addresses the traceability of requirements from the beginning of the development all the way down into the executable test code. The OMG standard SysML offers concepts to connect requirements with model elements that either satisfy or verify the related requirement. This enables Fokus!MBT to relate requirements to generated test cases fully automated. Finally, test reports can state what requirement was verified by what test case with what result and vice versa.

To achieve this, Fokus!MBT is able to import different requirements representations, in particular the Requirements Interchange Format (RIF) and SysML requirements. The combination of UTP and SysML allows the construction of powerful test models, directly defined in UML and converted to its corresponding TestingMM representation afterwards for further refinement.

3.2 Constraint-based Test Data Generation

Fokus!MBT provides a constraint-based test data generator that copes with the generation of both stimuli and expected oracle data for a given test case, called CoDOG. CoDOG is able to parse data constraints, either defined by the OCL or Java, which are subsequently passed to the included constraint solver. The parser itself is realized as extensible component. Parsers for further constraint languages are easily pluggable into the Fokus!MBT core component, so that the a tester may use any favored and suitable language for the description of constraints.

3.3 Test Code Generator

In order to execute the modeled test suites, Fokus!MBT offers a TTCN-3 exporter natively. Since Fokus!MBT comes with a test data generator, the resulting TTCN-3 code is completely executable, including the entire type system and data instances. Other languages are not supported yet, however, again the extensible architecture of Fokus!MBT allows an easy integration of further exporters.

Test Automation as a Model-Driven Engineering Process with MDTester

"Test Automation is Model-Driven Software Development"

Alain-G. Vouffo Feudjio

FOKUS (Fraunhofer Institute for Open Communication Systems)

Abstract. Recent progress in model-driven engineering (MDE) have made the prospective of applying the same type of approach for test automation yet more attractive. The model-driven testing (MDT) approach consists of designing platform-independent test (PIT) models at a high level of abstraction and transforming those automatically through platform-specific test (PST) models into executable test scripts. However, intelligent tool-support is a critical pre-condition to making that vision true. Beyond the fact that the process of modelling for test automation must be intuitive and efficient, it must also be ensured that the created test models are readable, valid and self-consistent. Furthermore, automated transformation of those models into executable test scripts and backwards are required to ensure round-trip engineering and a reuse of legacy test automation solutions. The MDTester is a collection of tools designed to address those issues. The test cases are designed using a graphical notation called UTML, that combines concepts of the UML Testing Profile (UTP) with known-patterns in test design.

1 Introduction

1.1 Architecture

The MDTester is based on the TOPCASED environment, which runs on the popular modular plugins architecture provided by the Eclipse IDE. This not only facilitates the integration of test design activities with general system design activities, but also ensures that the developed solutions can benefit from the large variety of proposed tools in that environment.

1.2 Methodology

MDTester guides test designers through the whole process from the analysis of system and user requirements, all the way down to executable test scripts. A large collection of wizards and guards are provided to ensure that errors and conceptual test design mistakes are avoided right-away. The result is a test design process that is simple and intuitive, but yet powerful enough to express complex

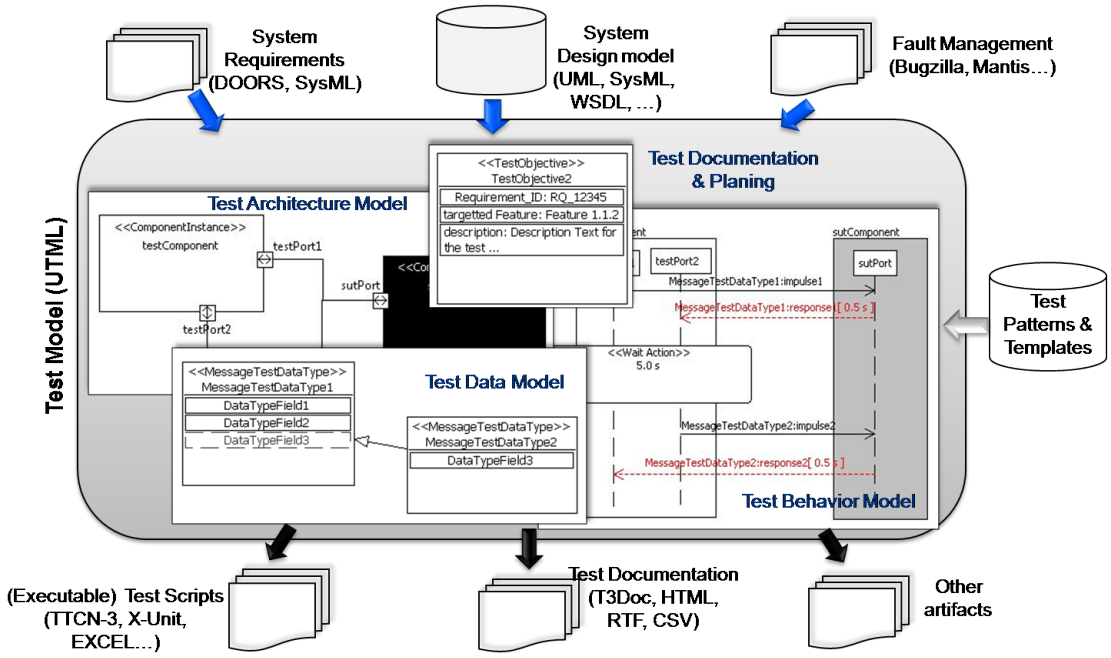


Fig. 1. MDTester Architecture

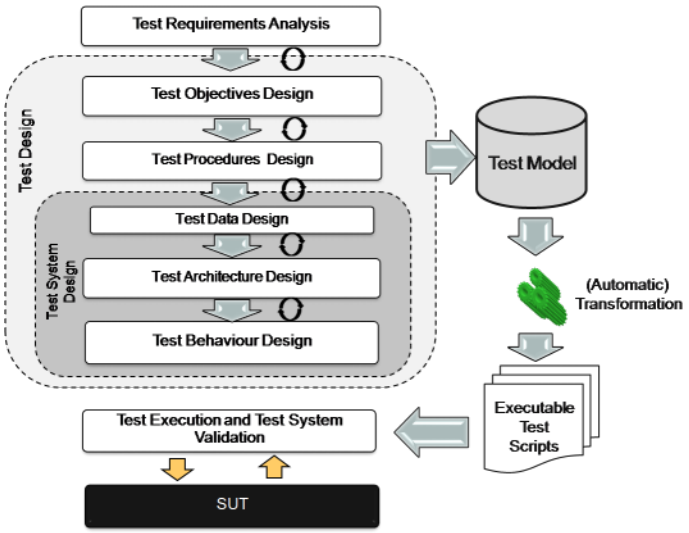


Fig. 2. MDTester Test Design Process

test behaviours. The UTML's graphical notation is simplified form of the UML notation, enhanced with specificities of test design. This ensures that the learning curve is kept low and makes it easy for people with less technical expertise, not just to understand test models, but also to design some themselves,without having to learn the details of a specific programming or scripting language.

2 Features

2.1 Test Design

MDTester supports all diagrams defined by the UTML notation to support the process depicted in Figure 2.

2.2 Test Model Validation

MDTester defines more than 50 built-in OCL-Validation rules that can be activated/deactivated as wished to verify that the created test models are valid and self-consistent. Additionally to those built-in rules, more validation rules can be added to the tool to perform quality assessment on the test models (e.g. based on company guidelines or similar)

2.3 Round-Trip Engineering

MDTester provides several back-ends and front-ends to allow round-trip engineering from PST models to executable test scripts and backwards. Additionally to the back-ends provided per default (TTCN-3, JUnit), MDTester defines an API through which customized back-ends and front-ends can be connected and used to generate proprietary test scripts or specific scripts appropriate to a given test execution environment

2.4 Requirements Traceability

SysML Requirements can be referenced in UTML test models via test objectives to provide traceability from test design to requirements and backwards.

2.5 Architecture Traceability

MDTester combines model-driven testing techniques with model-based automated test generation techniques (MBT) to transform system architectures into test architectures. This allows traceability of test cases back to the elements of the system architecture which they address and a quick identification of which test cases cover a given part of the system under test.

3 Contact Us

MDTester is available for download at `http://www.fokus.fraunhofer.de/go/utml` You may also contact us for more information via that site.

SIMOTEST: A Tool for Deploying Model-Based Testing in Matlab/Simulink® using IEEE 1641

Tanvir Hussain, Robert Eschbach, Martin Größl
{hussain | eschbach | groessl}@iese.fraunhofer.de
Fraunhofer IESE, Kaiserslautern, Germany

Introduction

Software is prevailing in most of the technical systems, even in the safety critical ones where hardware realizatio ns were once do minant. Th erefore, quality assuran ce of software i s ga ining e ver m ore im portance since i ts i nfluence i s gr owing l arger. I n most of t he d omains of a pplications s oftware i s bei ng de veloped u sing m odels, preferably usi ng exec utable m odels. They render the be nefits of abstractin g implementation t echnology rel ated fact ors and t hus rai sing i ts com prehensiveness, maintainability an d reu sability. Ex ecutable models ad ditionally provide th e advantages of verification and validation to reveal errors during the earlier phases of development. M atlab/Simulink® i s a po pular modelling and si mulation fram ework that p rovides ex tensive facilit ies fo r m odel based so ftware development an d simulative v alidation. However, th e m odels are o ften co mplex in stru cture and behavior and thus the verification and validation appears to be a daunting task. Model Based Test ing (M BT) e volved t o m ake ve rification a nd validation practicable and transparent.

The m odels use d i n M BT, o ften named as t est m odel, a re ab stract, part ial representations of t he Sy stem Under Test (SUT) an d ar e used f or generation an d selection of t est cases. But the test m odels ofte n cont ain abst ractions especially regarding th e in put and outputs. This as a result requires a m echanism for concretization of these elem ents since othe rwise execution of the test cases is not possible. In practice these c oncretizations are m ostly perform ed manually and are therefore o ften er ror pr one, pr oblem or p latform specific and can sel dom be reused . To s olve t his problem SI mulink M Odel-based TE STing (SIMOTEST) tool ca n be used. It resorts to IEEE 1 641 - stan dard fo r Sign al an d Test Defin ition (STD) th at provides the opportunity to describe signals and signal measurements of wide variants in a modular and unique way. This article presents a short overview of STD as well as SIMOTEST.

IEEE 1641 – standard for Signal and Test Definition (STD)

IEEE 1641 standard is organized into following four layers:

1. **Signal Modeling Language (SML) Layer** can be used t o provide e xact mathematical definition of signals and signal measurements. The definitions can be formulated using the de-facto functional programming concepts of Haskell 98[1].

1 Haskell 98 Report: A Non-strict, Purely Functional Language

2. **Basic Signal Component (BSC) Layer** provides reusable, fundamental signal classes. BSCs can be formally defined through SML and provides the lowest level of signal building blocks available in STD.
3. **Test Signal Framework (TSF) Layer** identifies how libraries of signal definitions can be used. TSF is the extendibility mechanism where parameterization and composition is applied. BSCs can be parameterized and/or composed to form TSFs as well as resultant TSFs can also be used in the mechanism to formulate other TSFs.
4. **Test Procedure Language (TPL) Layer** provides the capabilities to define test descriptions composed of test requirements, i.e., initialization, software or component requirements etc., and test procedures, i.e., causal chain of relations between the signals and measurements etc.

SIMOTEST

SIMOTEST provides necessary support for formulating STD compliant test descriptions that can be used efficiently to deploy MBT. The artefacts of MBT and STD can be combined together in SIMOTEST to finally obtain an executable description of the test where each test case is augmented with necessary initialization as well as the concrete signal and measurement descriptions corresponding to test stimulation and evaluation. Additionally, SIMOTEST provides enough automation for configuring an executable test in Matlab/Simulink®. Due to the unique and portable definition of the modules in STD it is possible to reuse the test descriptions also for testing code generated from the model as well as the resulting system.

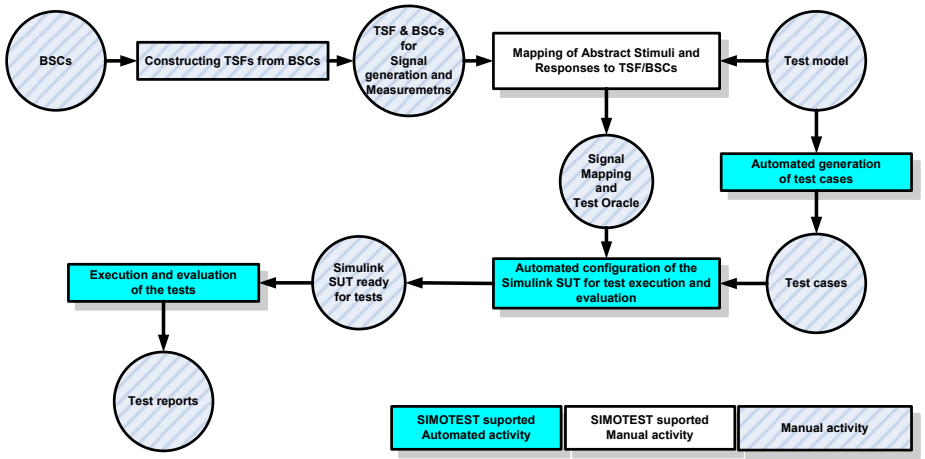


Figure 1: Workflow for deploying MBT using SIMOTEST

Figure 1 shows the workflow for deploying MBT using SIMOTEST. Inspired by a case study[2] in the research project D-Mint[3] a demonstrator for phase angle control was developed. During the development cycle the hardware which generates these signals is sometimes not available or in development thus a realistic test of the software should have to be performed with simulated signals.

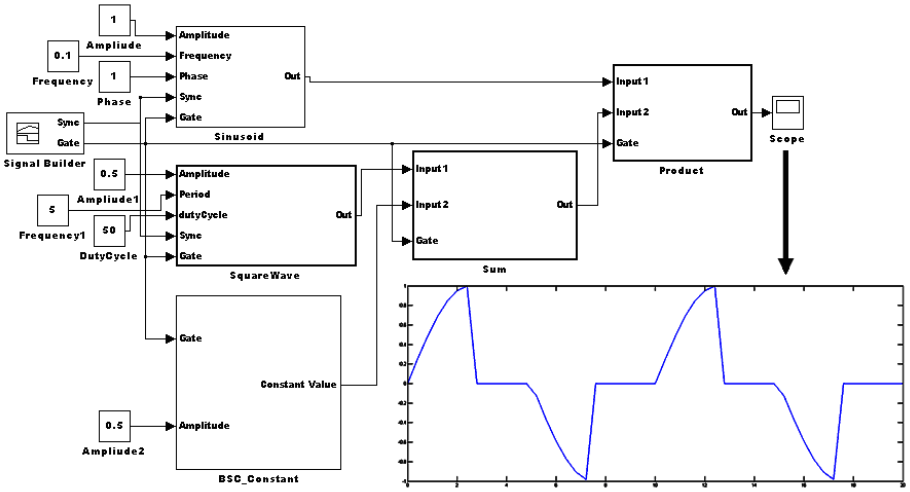


Figure 2: Generation of pulse width modulated sinusoidal signal using STD

Figure 2 shows a STD compliant generation of pulse width modulated signals for testing the model of the soft starter software.

2 Bauer, T., Eschbach, R., Größl, M., Hussain, T., Streitferdt, D., and Kantz, F. Combining combinatorial and model-based test approaches for highly configurable safety-critical systems. In Proc. of MoTiP, pp. 9 – 22.

3 http://www.d-mint.org/

SESSION 3
Commercial model-based test tools

smartesting®
Optimize your Test Center

Test Designer™ 4.0

Product Datasheet

Automate your test design with Model-Based Testing

Traditionally, we think of testers as the writers of test scripts through a long and tedious manual process.

This manual testing process typically extends from the initial steps when assessing test requirements, to the final step when successful test verdicts are obtained. For mature test practitioners Application Lifecycle Management implies robust iterative software delivery from design to production delivery. Those testers want to be productive and predictable while delivering both reliable and systematic functional validation.

The Smartesting Center™ solution automates the test design step and integrates Test Designer™ with requirement management, UML modeling and test management solutions. Thanks to the benefits of automated test design you can start now to implement a fully industrialized tool chain based on your current testing tools. Test Designer™ implements built-in integrations with Borland, HP, IBM solutions and creates additional value on top your current practices.

What if design and related specifications change?

The amount of effort required for each new iteration (when some change occurs) is often difficult to estimate and adds risks to projects. Consequently, even if functional testing and test automation are well established practices with easy to find guidelines, expert resources and commercial-off-the-shelf products, the expected cost of testing is typically far from being predictive. This statement illustrates test practitioner's needs for reducing risks and improving drastically their current testing practices.

'Witness Test Designer™ high productivity engine for test cases and test scripts generation from models'

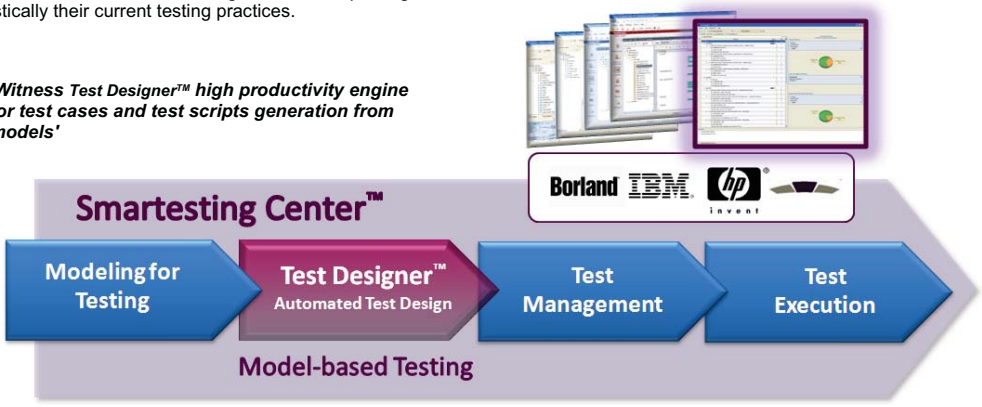
Smartesting Center™ with Test Designer™ Key Value Points

Only Test Designer™ provides **automation** between formal description of the **business needs** captured by Analysts, and repeatable **regression testing assets** manipulated by Testers.

Only Test Designer™ provides built in **integration to existing tools**, methodologies and processes for requirement management, UML modeling, and test management.

Only Smartesting Center™ extends **Model-Based testing** concept to its actual implementation though the **Test Designer™ engine** that provides 'executable specifications' from UML models.

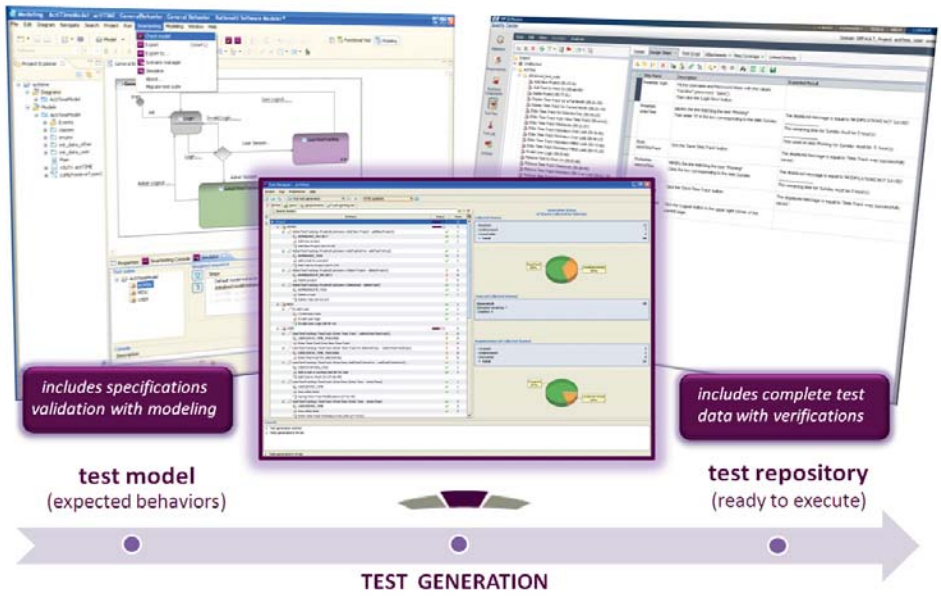
Only Smartesting Center™ makes the **Test Factory** industrialization concept real with its unique and patented productive capabilities.



smartesting®
Optimize your Test Center

Test Designer™ 4.0

Product Datasheet



Test Designer™ 4.0 integrations

Supported UML Modelers

- IBM Rational Software Modeler v7.0.5.x or v7.5.x
- IBM Rational Software Architect v7.5.x
- Borland Together 2008

Supported Test Management Environments

- Mercury Test Director 8.0, HP Quality Center 9.x
- HP QuickTest Professional 8.2 to 9.x
- QTP plug-in for Quality Center

Third-party test environments supported on request

Test Designer™ 4.0 system requirements

Operating System recommendations

- MS Windows 2000, XP Pro SP2, Server, Vista SP1, Business, Enterprise, and Ultimate Edition
- Ubuntu GNU/Linux distribution (Desktop Edition)
- Java Runtime Environment: v1.6.0 or higher

Minimum RAM: 1GB, 2 GB recommended

Disk Space: 70 MB free space required

CPU: 1GHz Single Core, 2.8GHz recommended

Key features include:

• UML Model-driven formal description and simulation of expected behaviors and test objectives,

• Maintain traceability links between requirements, models and generated tests,

• Expedites adoption within third party test management tools of iterative testing, shortens test design cycles when requirements change,

• Graphical user-driven generation of executable test scripts from UML directly to your favorite Test Management environment.

Smartesting® is a referenced business partner of HP®, Borland® and IBM®

contact us at www.smartesting.com

QTRONIC™

WOULD YOU RATHER MODEL THE TESTS OR THE SYSTEM UNDER TEST?

Model-based testing (MBT) is recognized as an effective means of addressing the exponentially growing demands on test design for software and systems. Most MBT approaches provide tools for capturing test scripts at a higher level of abstraction. A more advanced approach to MBT is focusing on modeling the requirements and behavior of the system under test and relies on automated test design tools to derive the proper test plans and test scripts. The challenge of developing automated test design tools has made them the subject of academic research and internal efforts within corporations — until now!

Conformiq Qtronic™ is a commercially available, automated test design tool that uses a system requirements model as its input to automatically generate test scripts and test plan. Conformiq Qtronic seamlessly integrates with your existing test execution environment while the requirements model serves as an executable specification that improves the communication between the product development and test design teams.

DELIVERING THE BENEFITS OF MBT, NOW!

Conformiq now delivers the industry-leading, commercially proven, automated test design tool Conformiq Qtronic to help you realize the following benefits:

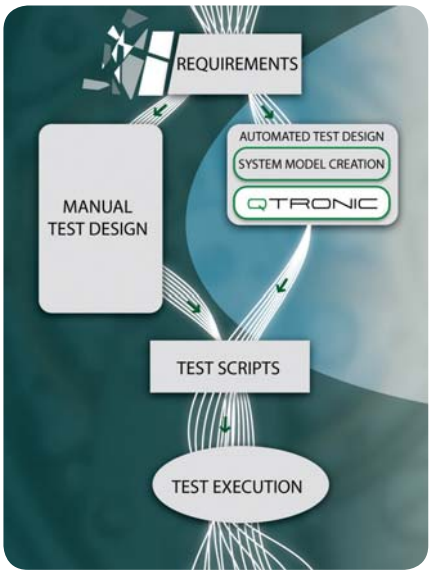
Faster Test Design: System model creation requires only a fraction of the time consumed in manual test script design. Automated test script generation has been shown to be 2 to 20 times faster than manual script design. The system model is used to also improve communication between the design and test teams, resulting in further time to market reductions.

Higher Test Quality: Conformiq Qtronic eliminates the risk of accidentally missing or incorrectly designed test cases, resulting in higher test quality and reliability.

Better Test Coverage: Conformiq Qtronic offers user controls for generating various levels of requirements coverage. The automatically generated test plan documentation clearly identifies which test cases cover what requirements.

Easier Test Maintenance: During the product development, requirements usually evolve, and test plans must be updated to reflect the new requirements. It is much easier to update the system requirements model as opposed to updating the test scripts.

More Test Reuse: Working with the requirements model versus the test scripts, is similar to working with a high level programming language versus assembly code. Requirements models can be more easily understood and re-used from one project to the next, in whole or in parts.



THE QTRONIC™ ADVANTAGE

Ease of model import: From third party modeling tools or from Conformiq Modeler™, in either graphical or textual form using industry standard object oriented language.

Supports Incremental Model Creation: Start with a simple model and augment it as product requirements change. Join multiple models to create higher level models.

Automatic Test Case Generation: The tool automatically identifies a number of test cases that together cover the requirements selected for test generation.

Calculates and Generates Test Data: Appropriate test input data is automatically calculated and generated by the tool without any further input from the user.

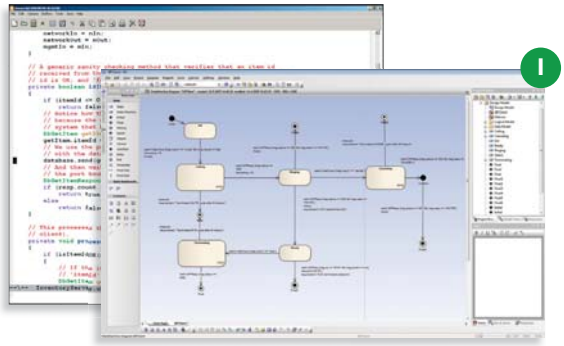
Test Result Evaluation: Correct expected outputs are generated by the tool completely automatically.

Ease of Integration with Existing Test Automation Tools: Plug-ins and open API's enable the creation of custom output formats and seamless integration with your test environment.

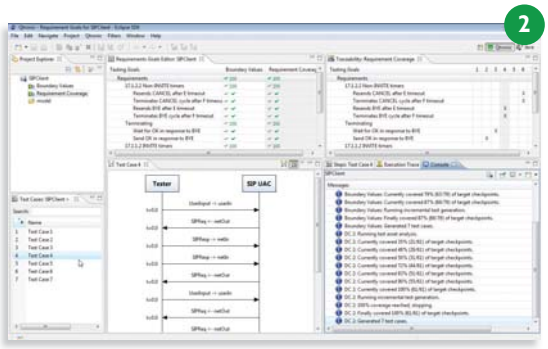
www.conformiq.com conformiq@conformiq.com

CONFORMIQ

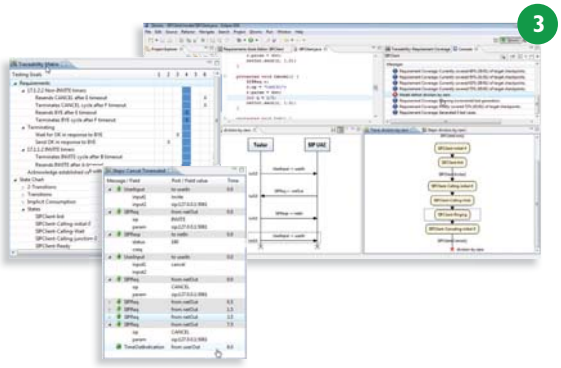
Automated Test Design™



Create requirements models in UML and Java-compatible notation.



Automatically generate tests by simply pressing a single button. Review and analyze the generated test cases in the user interface using message sequence charts and test execution traces.



Review and analyze test design results using automatically generated traceability matrix. Do detailed analysis of the test steps with test data and timings. Debug the model for errors and defects. All in a single working environment.

AUTOMATED TEST DESIGN FLOW

Conformiq Qtronic™ is designed to seamlessly integrate in your existing test environment and follows a three step process:

1 **Requirements Model Creation:** A combination of UML state charts and Java-compatible code are used to create the requirements model. Third party UML modeling tools can be used for the creation of the model. A lightweight modeling tool, the Conformiq Modeler™ is also included with Conformiq Qtronic.

2 **Automatic Test Generation:** The requirements model is used as input to the Conformiq Qtronic automated test generation engine. User controls enable selection of requirements to be tested and of the level of completeness of the generated test cases. Conformiq Qtronic automatically generates the executable test scripts and associated test plan documentation—including traceability matrices for requirements and state transitions, message sequence charts, etc.

3 **Integration with Test Execution:** Executable test scripts can be generated in a variety of formats—C/C++, Java, TCL, Perl, Python, TTCN-3 —to seamlessly integrate with the test harness and test execution environment you are using. You can also create yourself, or contract Conformiq C2S2™ services to create, plug-ins that generate desired output formats.

Conformiq Qtronic is available as Eclipse® plug-in for Microsoft® Windows® 2000/XP/Vista, Linux and Solaris.

CONFORMIQ CUSTOMER SUCCESS SERVICES: C2S2™ METHODOLOGY

Automated Test Design represents a paradigm shift in test design for software and systems. C2S2 methodology has been developed with one objective in mind: To help your successful adoption of Automated Test Design in your test process. We offer product training, integration, and project execution services. Our approach results in a winning combination of your domain knowledge and our automated test design experience.

Step into the future of test design, Contact us.

conformiq@conformiq.com
www.conformiq.com

12930 Saratoga Av. Suite B9
Saratoga, CA 95070
USA
Tel: +1 408 898 2140
Fax: +1 408 725 8405

Westendintie 1
02160 Espoo
Finland
Tel: +358 10 286 6300
Fax: +358 10 286 6309

Norrmalmstorg 14
SE-111 46 Stockholm
Sweden
Tel: +46 852 507 094
Fax: +358 10 286 6309