TUCS

Qaisar Ahmad Malik

Combining Model-Based Testing and Stepwise Formal Development

Turku Centre *for* Computer Science

TUCS Dissertations

No 130, September 2010

Combining Model-Based Testing and Stepwise Formal Development

Qaisar Ahmad Malik

To be presented, with the permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Gamma, the ICT building, on October 4th, 2010, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5
20520 Turku
Finland

2010

Supervisor

Professor Johan Lilius
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5
20520 Turku
Finland

Reviewers

Professor Tommi Mikkonen
Department of Software Systems
Tampere University of Technology
Korkeakoulunkatu 1
FIN-33101 Tampere
Finland

Adjunct Professor Ville Leppänen
Department of Information Technology
University of Turku
Joukahaisenkatu 3-5
FIN-20520 Turku
Finland

Opponent

Professor Tommi Mikkonen
Department of Software Systems
Tampere University of Technology
Korkeakoulunkatu 1
FIN-33101 Tampere
Finland

ISBN 978-952-12-2467-6
ISSN 1239-1883

To the memory of my beloved father, Mohammad Latif Anwar (1930-1994)

Sammanfattning

Datorer har blivit en nödvändighet i vårt dagliga liv. De används i en mängd olika system, allt från avancerade system såsom flygplan, telekommunikation och banksystem till hushållsapparater såsom TV, tvättmaskiner och mikrovågsugnar. Programvaran spelar en viktig roll i många av dessa system. Med framstegen inom teknik har datorer och i synnerhet deras programvara blivit ganska komplicerade. Eftersom vårt beroende av programvara ständigt ökar, förväntar vi oss inte att programvaran ska fallera. Programvara har dock ofta fel. Resultatet av programvarufel kan vara allt från mindre irritationsmoment till stora ekonomiska förluster eller katastrofala situationer.

Fel i programvara kan bero på en felaktig tolkning av krav, designfel eller slarviga misstag av programmerare. Det finns flera sätt att kontrollera kvaliteten hos programvara, inklusive rigorös och noggrann analys, design samt styrning av utvecklingsprocessen. Testning av programvaran kvarstår dock fortfarande som den mest använda metoden inom industrin för kvalitetssäkring. En effektiv och omfattande testning av programvaran är en mödosam, tidskrävande och felbenägen process i synnerhet eftersom den ofta görs manuellt. En avancerad metod för testning är modellbaserad testning (MBT) där testfall genereras från modeller av systemet. Modeller som används för detta ändamål är formella eller semiformella modeller med exakt semantik. Dessa modeller erhålls vanligen genom att följa vissa formella utvecklingsmetoder. De formella utvecklingsmetoderna garanterar riktigheten av ett system genom att tillämpa metoder från matematik och logik. Den forskning som presenteras i denna avhandling kombinerar den formella mjukvaruutvecklingsmetoden Event-B med MBT. För MBT använder vi scenariobaserade tester som grund för våra testvalskriterier.

Avhandlingen är uppdelad i två delar. Den första delen ger bakgrundsinformation om ämnet och sammanfattar våra forskningsbidrag. Den andra delen av avhandlingen, som består av ursprungliga publikationer, diskuterar hur stegvis formell utveckling, även kallad precisering, kan användas för att härleda testfall. Publikation I presenterar en algoritm för att härleda scenariobaserade testfall genom en serie förbättringar. Idén om scenariobaserade tester utökas i Publikation II och en alternativ metod för förädling av sce-

narier föreslås med hjälp av en fallstudie. Publikation III presenterar en mekanism för att omvandla scenariobaserade abstrakta testfall till JUnit tester, som exekveras mot Java implementationen av systemet-under-test. Under den beskrivna processen överförs kraven som är kopplade till testscenarier till JUnit test. I publikation IV har vi använt UML och UML-B modeller för MBT, följande en stegvis utvecklings strategi. För testgeneration använde vi Conformiqs Qtronic verktyg. Publikation V behandlar en jämförande studie av två modelleringsperspektiv som används för modellbaserad testning. Denna analys görs på basis av två fallstudier från telekommunikationsdomänen.

Acknowledgments

First praise is to ALLAH, the Almighty, on whom ultimately we depend for sustenance and guidance.

I am glad that this day has arrived when I am writing this part of my thesis. This thesis is not a product of efforts of a single person and it is my pleasure to show my gratitude to all those who made this thesis possible.

I am heartily thankful to my supervisor, Professor Johan Lilius, whose encouragement, guidance, support and patience made this thesis come true. I would also like to thank him for his friendly attitude and fair advices on various matters during my doctoral studies.

I would also like to thank my thesis reviewers, Professor Tommi Mikkonen from Tampere University of Technology and Adjunct Professor Ville Leppänen from University of Turku, for their time and efforts while reviewing the thesis and providing with their invaluable comments and suggestions how to improve this work. Furthermore, I wish to sincerely thank Professor Tommi Mikkonen for accepting to be the opponent in the public defense of my thesis.

I owe my deepest gratitude to my senior colleagues and friends, Dr. Dragoş Truşcan and Dr. Linas Laibinis. You guys were always there to guide me and help me in resolving difficult research issues, often on a very short time notice. Just as you were there for me to help me finalize and improve my papers. Thank you very much for doing that!

I would also like to extend my gratitude to all of my co-authors for collaborating on this research. Special thanks go to post-doctoral researcher Dr. Manoranjan Satpathy (now at General Motors, India) as well as Dr. Mika Katara and his model-based testing group from Tampere University of Technology for their collaboration and guidance during our joint work.

I would also like to take this opportunity to thank Professor Kaisa Sere for accepting me as her doctoral student at the Distributed Systems Laboratory. Although I later moved to the Embedded Systems Laboratory, to join the D-MINT project under the supervision of Professor Johan Lilius, I could however always feel her support and encouragement for my success. I really appreciate her trust in me. During my stay at the Dis-

tributed Systems Laboratory, I learnt a great deal from senior researchers Marina Waldén, Elena Troubitsyna and Luigia Petre. Moreover, there I also made many good friends. In particular, I am very thankful to Pontus Boström, Leonidas Tsiopoulos, Mats Neovius, Marta Olszewska, Dubravka Ilić, Fredrik Degerlund and Anton Tarasyuk for their invaluable friendship.

My stay at the Embedded Systems Laboratory was made enjoyable due to the pleasant and amusing working environment of this lab. It always helped me to stay in good mood, regardless of all work-related stress and worries. In this regard, I am grateful to all my fellow workers and friends especially to Johan Ersfolk, Mohsin Saleemi, Kristian Nybom, Haitham Habli, Sébastien Lafond, Andreas Dahlin and Stefan Grönroos.

I am blessed to have made many friends at my workplace and I want to extend my gratitude to all other friends and colleagues from the Department of Information Technologies at Åbo Akademi and Turku Centre for Computer Science (TUCS), especially Moazzam Niazi, Ali Hanzala, Mikolaj Olszewski, Adnan Ashraf, Kashif Javed, Fredrik Abbors and also to all others for exchanging smiles and saying "Hi" while passing through the department corridors.

I have greatly enjoyed the study and research environment provided by TUCS and the Department of Information Technologies. I am grateful to both of these institutions for providing generous financial support for my doctoral studies. All of my research and conference trips were financed either by TUCS or by European Union (EU) funded projects namely, RODIN (Rigorous Open Development Environment for Complex Systems) and D-MINT (Deployment of Model-based Technologies to Industrial Testing). Therefore, I would like to extend my gratitude to EU and to everyone who made these projects possible. I would also like to thank all secretaries, administrative and support staff at TUCS and Åbo Akademi for running the department in an efficient manner and keeping it free from hectic bureaucratic procedures. In this regard, I am glad to name Christel Engblom, Irmeli Laine, Britt-Marie Villstrand and Ulla Bäckström for their kind help at different phases of my doctoral research.

Last, but certainly not least, I wish to thank my family for all their wishes, prayers and love they gave me. I am truly indebted to the support of my parents, especially of my mother, who after death of my father and regardless of major financial difficulties, took special care of my education. Without her help, encouragement, prayers and unconditional love, I would not have managed to come to this stage in my life. Thank you very much Maa Jee!

Finally, I owe my loving thanks to my wife and my best friend, Kanwal, for standing by me at all times throughout this journey. Without your great sense of humor, wit and love, my life would have been very boring. Thank you for everything!

List of Original Publications

This thesis is based on five original papers, which are referred in the text by Roman numbers I-V.

I Manoranjan Satpathy, Qaisar A. Malik and Johan Lilius. Synthesis of Scenario Based Test Cases from B Models, *In Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification (FATES/RV)*. Lecture Notes in Computer Science, Vol. 4262/2006, pp. 133-147, Springer-Verlag, August 2006, Seattle USA.

II Qaisar A. Malik, Johan Lilius and Linas Laibinis. Model-based Testing Using Scenarios and Event-B Refinements, *In Methods, Models and Tools for Fault Tolerance*. Lecture Notes in Computer Science, Vol. 5454/2009, pp. 177-195, Springer-Verlag, March 2009.

III Qaisar A. Malik, Johan Lilius, Linas Laibinis and Dragoş Truşcan. On Extending Scenario-based Test Case Generation Using Event-B Models, (Published with additions as "Requirement-driven Scenario-based Testing Using Formal Stepwise Development" in *International Journal On Advances in Software*. Vol. 3 Nr. 1 & 2, pp. 147-160, 2010.)

IV Qaisar A. Malik, Dragoş Truşcan and Johan Lilius. Using UML Models and Formal Verification in Model-Based Testing. *In Proceedings of 17th IEEE Intl. Conference on Engineering of Computer-Based Systems (ECBS 2010)*, IEEE Computer Society, pp. 50-56, March 2010, Oxford, UK.

V Qaisar A. Malik, Antti Jääskeläinen, Heikki Virtanen, Mika Katara, Fredrik Abbors, Dragoş Truşcan and Johan Lilius. Model-Based Testing using System vs. Test Models -What is the difference? *In Proceedings of 17th IEEE Intl. Conference on Engineering of Computer-Based Systems (ECBS 2010)*, IEEE Computer Society, pp. 291-299, March 2010, Oxford, UK.

"Seek knowledge from the cradle to the grave."
Prophet Muhammad (Peace be upon him)

Contents

Part I

Research Summary

1 Introduction

Computers have become a necessity of our everyday lives. They are used in a variety of systems, ranging from sophisticated ones such as aircrafts, telecommunications and banking systems to home appliances such as TV, washing machines and microwave ovens. Software plays an important role in many of these systems. As a result, the usage of software for a variety of purposes in different domains of modern life is rapidly increasing. With the advancements in technology, computers and in particular their software have become quite complex. Due to this reason, the correctness of software cannot be guaranteed, even by the programmer who has designed it.

Since our dependence on software is continuously increasing, we do not expect it to fail. However, software often contains errors. The severity of software errors can range from minor irritations to major economical losses or catastrophic situations. An often quoted example is of the software error that caused the failure of Arian 5 Flight 501 [36], resulting in the loss of more than US$370 million and several years of research. The increased penetration of software-intensive systems in our society has also increased the demand for high-quality software. Moreover, due to sever market competition, it has become a matter of reputation for organizations to produce quality products, especially when safety of people or environment is concerned. A recent example of such a case is the Toyota company, which issued a recall of approximately 133,000 Prius and 14,500 Lexus vehicles to update software in the vehicle's antilock braking system [75].

The errors in software can be due to misinterpreted requirements, design violations or careless mistakes done by programmers. There are several ways to control the quality of a software, including rigorous and careful analysis, design and process management. However, software testing still remains the primary method used in industry for quality assurance. Software testing can be described as an activity of checking whether for given inputs the software always produces the correct output. Software testing has mostly been a manual job, which usually involves a feedback loop between developers and testers. Once a software module is developed and sent for testing, the testers apply the pre-defined procedures to find and report errors. In response, the developers fix them and send the software again for testing. This feedback loop continues until all found errors are resolved.

Effective and extensive testing of a software system is a laborious, time-consuming and error-prone process, especially since it is often performed manually. This can be improved by automating the execution process. Traditionally, test automation is supported by pre-defined scripts, which generate test inputs, and then execute test cases on the system under test (SUT). However, the automation does not distinguish between less and more important (e.g., critical) parts of the system. Moreover, exhaustive testing by

automation is not feasible since generating and running tests covering all possible values of data or execution paths of a program is too difficult and time consuming.

An advanced approach to testing is model-based testing (MBT) [85] where test cases are generated from existing models of the system. The models used for this purpose are formal or semi-formal models with precise semantics. Very often, these models are also used to generate input data and expected output values for the test cases. The research work presented in this thesis is based on MBT.

Another approach for quality assurance is formal methods for software construction. The formal methods prove correctness of a system by applying techniques from mathematics and logic. The software developed using formal methods is considered correct-by-construction with regard to formal system models (specifications), hence, theoretically speaking, needs no testing. However, the formalists encourage testing as it helps in detecting errors and omissions of the specifications. The use of formal methods in industry has been limited by a number of factors, especially by its still limited applicability for the development of large sized software. However, the formal methods provide a number of specification and verification techniques that can be used at least at the design level.

In the past, formal methods and testing were often considered as rival approaches. However, during recent years, the researchers have repeatedly tried to combine them in order to get the best of two worlds. This combination has advanced especially in the context of MBT, where formal models are used to generate test cases. This thesis is also a contribution in this direction.

1.1 Research problems

In the following, a list of research problems, which have been addressed in this thesis, and their proposed solutions are outlined.

- *Using formal methods in order to improve the quality of the models used for MBT*

  We use formal models for modeling and verifying the system behavior at an abstract level. The verified models are then used in our MBT approach for generating tests (Publications I to IV). Combining formal methods and MBT is not a brand-new concept. However, our work in this direction is focused on using formal stepwise development, also known as *refinement*. In our proposed approach, the test design follows the same refinement steps used for developing formal specifications. We use the Event-B [10, 11] formalism for formal stepwise

development. The models are mathematically proved as correct before they are used for MBT.

- *Using user-provided abstract testing scenarios as test selection criteria*

  In MBT, there is often a nearly infinite number of tests that can be generated. Even if a finite, but large, number of tests are generated, it is usually not feasible to execute all of them due to restrictions on available time and resources. In this regard, one needs to use some selection criteria to focus on particular parts or aspects of the system that must be tested. The selection criteria we use in this thesis (Publications I to III) are based on the testing scenarios that are provided by the user. The scenarios represent the system behavior to be tested at an abstract level. Moreover, these scenarios are refined in a step-wise manner following the same refinement steps used for refining formal models of the system.

- *Transforming abstract test cases into executable ones*

  In MBT, the models used to represent the system-under-test (SUT) are usually abstract. Hence, the tests generated from these models are also abstract. In order to execute these tests on the target system, the tests need to be in an executable form with all the necessary details like inputs, the sequence of steps to follow, and the expected outputs. In this thesis, we present a methodology (in Publication III) to transform abstract test cases into executable Java Unit (JUnit) [5] test cases.

- *Using UML and UML-B models for test generation*

  In a separate work (Publication IV) we have explored using UML [71] and UML-B [79] models for MBT while following the stepwise development approach. The behavior of the SUT is modeled in UML-B, while the architectural parts are modeled in UML. The main advantage of using UML-B is that it provides a graphical modeling facility combining constructs from both UML and Event-B. The resulting models are then exported to Conformiq's Qtronic [55] for test generation.

- *Comparing two modeling perspectives used in MBT*

  Finally, in the last part of the thesis (Publication V) we compare, by presenting two case study examples, two modeling perspectives used in MBT, namely, modeling the SUT and modeling the environment of the SUT. This is done by comparative analysis of two case study examples from the telecommunication domain.

1.2 Research methodology

The main research methodology used in this thesis has both exploratory and analytical aspects. We started with studying the advantages and disadvantages of using the Event-B formalism for model-based testing. Later on, we devised a methodology for refining the test cases along with the formal development (refinement) steps. As a first step in this direction, we devised an algorithm, presented in Publication I, for refining the scenario-based test cases. This algorithm proved to be exponential in nature, thus limiting its applicability. In Publication II, we resolved the exponential problem of the first publication by introducing another scenario-based testing methodology. This new methodology was quite different than the first one as it involved an additional technique for checking conformance between the scenarios and the model by ProB model-checker [62]. In Publication III, the scenario-based testing approach was further extended as we devised a technique to transform abstract test cases into executable ones. Furthermore, we also introduced a requirement traceability feature in our methodology. In short, the above described research work on scenario-based testing evolved with the time, while improving the existing features and exploring the required future extensions. In another model-based testing work, presented in Publication IV, we explored using UML and UML-B to improve quality of the models used for test generation. For this work, we adapted an existing MBT methodology, named MATERA [8], to use UML-B models while following the formal stepwise development approach. In Publication V, we analytically compared two modeling perspectives used in the MBT, namely, modeling the SUT and modeling the environment of the SUT.

For demonstrating the proof-of-concept in our publications we worked on various case study examples, which also involved experimentation with various software tools and languages, e.g., the Rodin [7] platform and supported provers, the ProB [62] model-checker, the JUnit framework [5], EclEmma [3], UML [71], UML-B [79] and Qtronic [55]. Overall, the analytical aspect of our research nearly always came hand in hand with the exploratory one.

1.3 Research setting

The research presented in thesis has been done as a collaborative work at Turku Centre for Computer Science (TUCS). The research work was in the context of two research projects, namely, RODIN(Rigorous Open Development Environment for Complex Systems) [7] and D-MINT(Deployment of Model-based Technologies to Industrial Testing) [2]. The objective of the RODIN project was creation of a methodology and of a supporting open tool platform for the cost effective rigorous development of dependable complex software systems and services. This project was funded by the European

Union under the FP6 program and included several academic and industrial partners across the Europe. On the other hand, the D-MINT project aimed at the development, enhancement, and industrial deployment of testing methods and tools for software-intensive systems based on model-driven technologies. D-MINT, funded by ITEA2 [4], was a consortium of 27 partners both from industry and research. Last but not least, Publication V was a result of the collaboration with the researchers working on MBT at Tampere University of Technology (TUT), Finland.

1.4 Organization of the thesis

This thesis consists of two parts: Part I - Research Summary, and Part II - Original Publications. The Research Summary is structured a follows. Section 2 presents concepts of software testing and model-based testing. In Section 3, we describe a formal refinement-based development approach in general and the Event-B formalism in particular. Section 4 summarizes our main contributions and gives an overview of our approach and the related work for model-based testing using formal models. These contributions are based on the results of the publications listed on Page v. We continue with a short discussion on the use of formal methods and testing for the quality assurance in Section 5. Section 6 contains the summary and the conclusions.

2 Model-Based Testing

2.1 Software Testing

Software testing is the process of executing a program or system with an intent of finding errors [69]. It is considered one of the most important activities in the software development process. Studies [23, 39] show that testing typically takes more than 40% of the total development time. Usually, testing is used as a means of validation and verification. The distinction between the two is the following. The *validation* process describes whether the right software is built, while the *verification* describes if it is built right [14]. In other words, the validation process involves requirement documents which describe what the software is supposed to do, whereas, the verification process determines whether the software is correct with regard to its design.

In general, the software testing process cannot be exhaustive due to obvious limitations such as limited number of tests that can be performed in the given time. For this reason, the quotation of Dijkstra that *testing can be used to show the presence of errors, but never to show their absence* [34] has become a common statement. Hence, the testing aims at finding as many errors as possible. An *error*, or its synonym *bug*, can appear in many different forms, for example, as a logical, timing or user interface error. Just as there

are many types of errors, there are many different types of testing depending on the overall aim of the testing, for example, performance testing, security testing, requirement-based testing and functional testing. An exhaustive list on types of error and testing can be found in the literature, for example in the famous book on software testing techniques by Beizer [19].

Software testing can also be categorized based on the level of abstraction it is applied to. For example, according to the V-Model [12, page 6], as shown in Figure 1, the following testing types are defined.

- *Function/Unit testing* is applied to the smallest unit of the software program code.
- *Module testing* is used to test modules or components of the system.
- *Integration testing* is performed to assess the software with respect to its subsystem design.
- *System testing* is applied to test a complete system in order to assess the software with respect to its architectural design.
- *Acceptance testing* is performed to test whether a system satisfies the "acceptance criteria", i.e., its operational and business needs.

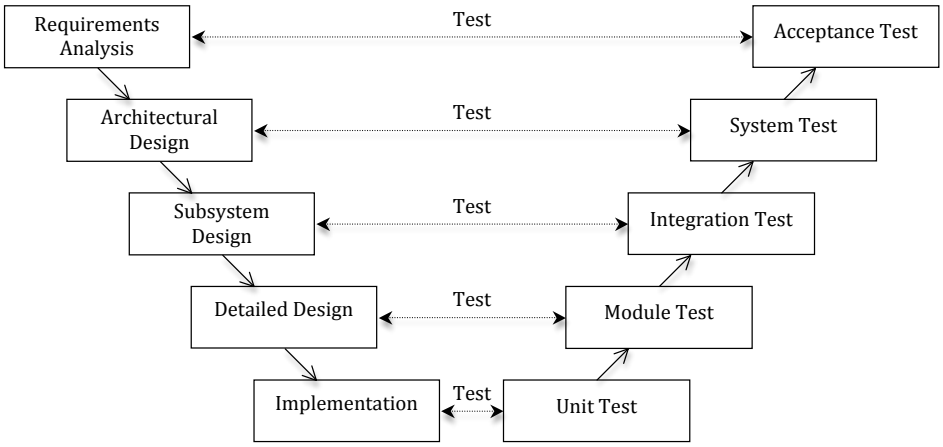


Figure 1: Software development activities and testing levels - the "V-Model"

Another often used categorization of software testing is based on the distinction between the *black-box* and *white-box testing*. In the *black-box testing*, a system-under-test is considered as a black-box, where internals of the system are not known. The only known parts are the inputs and the outputs of the system. This type of testing is also referred to as *functional*

testing. In the *white-box testing*, the testing is based on the actual source code of the software. The tests are designed in such a way that the test designer has access to the algorithms and structures of the source code. This type of testing is also referred to as *structural testing*. The test cases can execute certain parts of the source code, hence, it is possible to analyze the program code covered by the test cases and additionally, detect the dead-code that never got executed. Examples of the techniques used in white-box testing include fault-injection, mutation testing, static testing etc. Sometimes a hybrid approach called *grey-box testing* is used for creating functional tests while considering the internal structures of the program code.

2.2 Model-Based Testing Process

The term *model-based testing* is defined as a kind of testing where tests are generated from models [85]. The basic idea of model-based testing is the same as what was earlier known as specification-based testing [86]. The main difference between specification-based testing and model-based testing is that, in the latter, the model does not need to be a formal specification of the system. The model can merely be a representation of some aspects of the requirements to be tested.

In the following, the generic model-based testing process is briefly described. As shown in the Figure 2, the model-based testing process can be divided into five main steps (summarized from the book on *Practical Model-Based Testing* [85]).

The first step is referred to as *modeling* phase. A model can be of the system-under test (SUT) and/or of its environment. It is usually called an *abstract model* since it is less complex than the actual system implementation. The model is built from the requirements or the specification documents, and it encodes the intended behavior of the system. Since this model will later be used for test generation, this phase also involves a *test plan* to ensure that necessary requirements and design specifications have been considered.

The second step of model-based testing is *test case generation* from models. This process is tool-assisted (to mark this distinction, the box is drawn with bold line in the Figure 2). Since the test cases are generated from abstract models, they are on the same level of abstraction with the model, hence called *abstract test cases*. The abstract test cases focus on the key aspects to be tested and omit many other details. The test generation process often follows some *model coverage criteria* [12] and keeps track of requirements covered by the test cases.

The third step of model-based testing is to transform the abstract test cases, generated in the previous step, into executable concrete test cases.

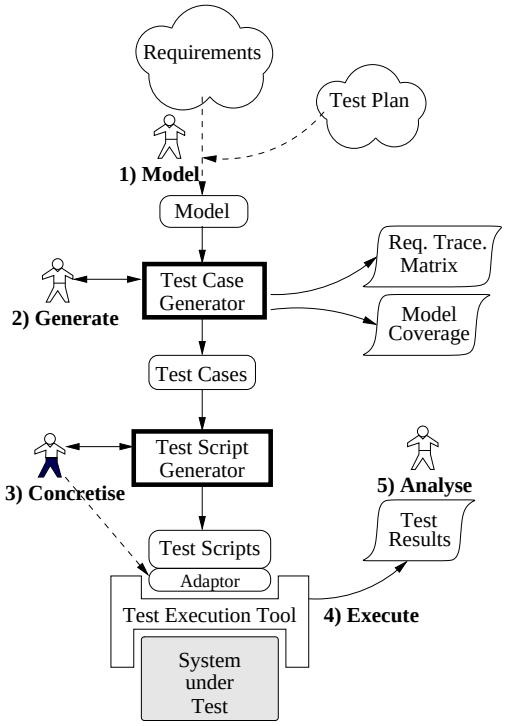


Figure 2: Model-Based Testing Process as in [85]

This is typically called *concretization* or *implementation* of tests. This process again requires tool support and/or is assisted by programmer(s). The transformation is carried out using various templates and mappings (between abstract and concrete test cases/values).

The fourth step is *test execution*. Before a test is executed, it needs to be adapted to the execution environment. This is performed by using a tool commonly referred to as *test adapter*. The adapter adds the implementation-specific information to the tests so that the tests can be executed against the real SUT. The tests can be executed either in *online* or in *offline* mode. In the online mode, the tests and inputs are applied as they are produced, on-the-fly based on the response of the SUT. Test execution and recording of the test results are managed by the provided model-based testing tool. On the other hand, in the offline mode, the concrete test cases are stored in the form of test scripts which are later executed manually or by using some tool.

The fifth step is the *analysis* of the test results. This phase is quite similar to traditional test analysis. Upon a test case failure, the analysis is performed to see whether the error is due to a fault in the system, in the test case itself or in the test setup.

2.3 Taxonomy of Model-Based Testing

In this section, we briefly describe model-based testing taxonomy presented by Utting, Pretschner and Legeard in [86]. The purpose of this description is to allow us later on, when we discuss our contributions in Section 4.5, to position them in the context of this taxonomy. This taxonomy is presented in Figure 3. It includes three general *classes*: model, test generation and test execution. Each class is further divided into *categories*. In the following we will give an overview of these classes and categories.

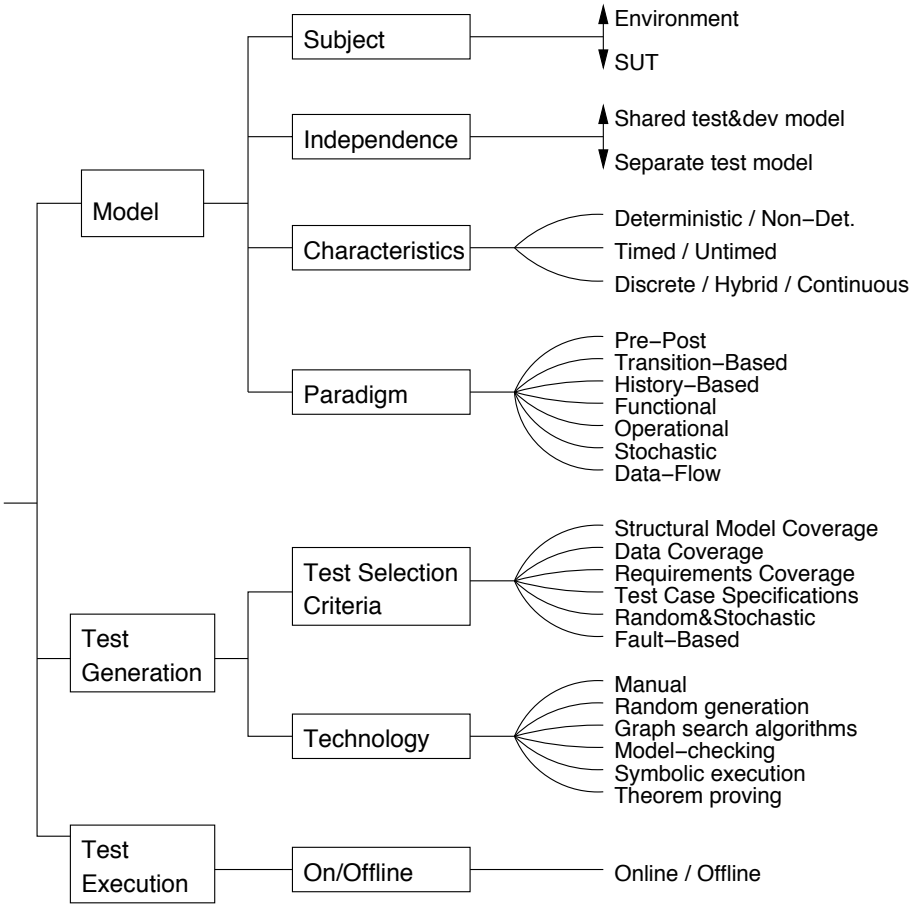


Figure 3: Model-Based Testing Taxonomy as in [86]

Model subject

The subject defines what is being modeled in the model-based testing process. The model can be a behavioral model of the SUT or of its environment/user interacting with the SUT. The difference between these two is in

the viewpoint with regards to the interfaces of the SUT. The models of the SUT provide an internal viewpoint by describing the inner architecture and reactions of the SUT to the external stimuli. On the other hand, the models of the environment provide an external viewpoint by describing what events the SUT should accept at a certain moment and by observing SUT reactions to these events. Sometimes, a combination of these above mentioned viewpoints is used. However, in that case the model becomes quite complex as it contains too much details to be practical [86].

Model Independence

A model used for model-based testing can be solely developed for this purpose or adapted from the development models. The development models, depending on the abstraction level, are used to represent inner details of the system and can be used to generate code. In order to save time and effort, sometimes it seems tempting to re-use the development models for the purpose of testing. However, if the same model is used for generating both the code and the test cases, then there is a chance that the mistakes in the model are propagated to both the code and the test cases, hence, reducing the capabilities of test cases to detect implementation errors. On the other hand, the models built solely for testing purposes, without any involvement of the developers of the SUT, are likely to reveal more implementation bugs and ambiguities in requirement specifications.

Model Characteristics

A model is an abstract representation of the SUT. Therefore, it exhibits some of the properties of the SUT while omitting other ones. The decision on what to include and what to leave out depends on the aim of the testing.

A model can be *deterministic* which means that, for a given input, the output of the model is deterministically defined. Sometimes, the SUT possesses concurrency due to internal parallel processes and then generates many alternate outputs depending on the current internal state, timing and hardware related asynchronous processes. The tests in this case are not represented as sequences anymore but rather as mathematical graph or tree structures.

Sometimes, there is a need to test the real-time constraints of the SUT. In this case, the models with the timing annotations are used. It is generally considered difficult to test such systems. In practice, simpler timeouts are programmed in the adapter than in the model.

In terms of dynamics, the models can be *discrete, continuous* or mixture of the two (also called *hybrid*). This characteristic is related to the type of the data used in the test cases.

Model Paradigm

The paradigm of model describes what style and notations are used for modeling. There are many different modeling notations and styles used in MBT these days. These are grouped into the following (adapted from van Lamsweerde [58]).

State-based (Pre/Post) notations: The system is modeled as a collection of variables, which provide a snapshot of the internal state of the system, and operations that modify the state. These operations are defined by a pre- and post-condition semantics. Examples of these notations include the B-method [9], Event-B [10, 11], Z [81], VDM [57] and JML [59].

Transition-based notations: The models using this type of notation describe transitions between different states of the system. Usually, the graphical node-and-arc notations, like finite state machine (FSM), are used to represent states (nodes) and transitions (arcs). Examples of these include FSM [89], State-Charts [49], Labelled Transition Systems (LTS) [82] and I/O automata [64].

History-based notations: These describe traces of the allowable system behavior over the time. Examples of these models include Message Sequence Charts (MSC) [29] and temporal logic [40].

Functional notations: Functional notations describe the system in the form of mathematical functions. Examples of these include first and higher-order logic specifications [13].

Operational notations: Models using these notations describe the system as a collection of parallel processes. These types of models are particularly suitable for describing distributed systems and communication protocols. Examples of such modeling notations include Process Algebras (e.g., CSP, CCS) [87] and Petri Nets [73].

Stochastic notations: The models using stochastic notations describe systems by the probabilities of events and input values. The Markov chain model [91] is one of the examples of such notation.

Data-flow notations: The models using this notation focus on the data rather than the control-flow of the system. Examples of data-flow notations include Lustre [45] and block-diagrams of Matlab Simulink [65].

Test Selection Criteria

This category defines how different tests are selected in the test generation process of MBT. In the following, various test selection criteria are briefly covered.

Structural Model Coverage: These criteria specify the selection of tests from models in terms of coverage of the structural elements of the models. Different modeling notations use different structural elements, therefore, the structural coverage criteria is also different in each case. Table 1 summarizes commonly used modeling notations and their respective coverage criteria examples.

Table 1: Modeling notations and Structural model coverage examples

Modeling Notation	**Structural Model Coverage Example**
State-based (Pre/Post)	Cause-Effect coverage, Disjunctive Normal Form (DNF) of post-conditions [12, page 138].
Transition-based	Various graph coverage criteria e.g., all-states, all-transitions, all-transition-pairs and all-cycles [12, page 32].
History-based	Each Message (EM) or Message Path (MP) coverage [63].
Functional	Coverage of axioms.
Operational	Similar to graph coverage e.g., transitions, places, cycles etc. [72].
Stochastic	Coverage from usage profile in terms of probabilities of states and transitions [90].
Data-flow	Data-path coverage, which is similar to code coverage criteria e.g., all-definitions, all-uses, all-definition-use pairs of data variables [41].

Data Coverage: These criteria describe how to select test-values from a huge data space. The most often used techniques include *boundary analysis*, where test input values are selected at the boundaries of the input domain, and *statistical distribution*, where input values are selected following certain statistical distribution [53].

Requirement Coverage: These coverage criteria are based on informal requirements of the system. The elements of the models, such as transitions of a state machine or predicates of a post-condition, are associated with the informal requirements. The generated test cases ensure that all the requirements of the SUT are covered.

Explicit Test-case specification Coverage: In this case, in addition to the model, the test cases are specified by the test engineer to emphasize testing of some particular aspects of the model. These test cases are often used to represent test objectives of the model, heavily used cases and scenarios. The test cases and models can be specified in the same or in different notations. Commonly used notations include FSM, UML Testing Profile (UTP) [43] and regular expressions.

Random and Stochastic Coverage: It is used for selecting tests based on the usage profile of the SUT in terms of action probabilities. A common approach is to use Markov chain models for this purpose because of their support for random variables and probabilities.

Fault-Based Coverage: This coverage criteria is used for testing typically occurring faults. An example of this criteria is mutation testing.

Test Generation Technology

Here we list various test generation technologies currently used in model-based testing.

Manual: In this case, tests are generated manually from the model.

Random Generation: It is carried out by sampling the input space of the SUT. This approach is also referred to as monkey-testing. Another common approach in this category is to generate tests by a random walk on the model.

Graph search algorithms: This technique uses graph-search algorithms to generate tests based on the coverage of the model as a graph. For example, the Chinese Postman algorithm [67] is used to test if each arc in a node-arc based graph is covered at least once.

Model-Checking: Model-checking [30] is used to verify the properties of a system using reachability analysis. The tests are specified as reachability properties, for example, *"eventually, a certain transition is fired or a certain state is reached"* (e.g., [52]). The model-checker generates a trace that eventually fires the given transition or reaches the given state.

Symbolic Execution: It is achieved by running an executable model with a set of input values as constraints (e.g., [74]). The resulting traces are then instantiated with concrete values to construct test cases.

Theorem Proving: Theorem provers [37] are used for test generation in a similar way as model-checkers are used. The theorem-provers check the feasibility of formulas that appear as the guards of transitions in state-based models.

Test Execution

Test execution concerns with the relative timing of test case generation and execution. In *online* testing, the test generation algorithms interact with the SUT and respond to the output of the SUT. This kind of testing is usually performed when testing non-deterministic systems. The test generator can see what path the SUT has followed and it follows the same path in the model. This is often referred to as *on-the-fly* testing.

In *offline* testing, the tests are strictly generated before they are executed. Usually, the generated tests represent linear traces and therefore are suitable for testing deterministic systems. Once generated, these tests can be reused in the future, for instance for regression testing.

2.4 Scenario-Based Testing

In the previous section, we have discussed various test selection techniques. One of the important test selection techniques among these is testing against the use cases of the SUT. Here, we are emphasizing on this testing technique because most of the later parts of this thesis deal with this technique.

A *use case* represents interaction(s) between the system and its user. Informally, the use cases are defined as: *description of sequences of events that, taken together, lead to a system doing something useful* [20, page 2-3]. The use cases are also referred to as *usage scenarios* or simply as *scenarios*. Hence, the testing based on such scenarios is referred to as *scenario-based testing*.

The scenarios specify the test cases on abstract level. They provide a way to express user expectations about the SUT which might be hard to test by other means. For instance, a scenario may include some important functionality of the SUT that might have been missed, mistakenly, in the model. Hence, the traditional coverage criteria, like code coverage, transition coverage etc., do not guarantee that the user scenarios are tested. Therefore, the scenario testing provides another way of ensuring that the user requirements and expectations about the SUT have been fulfilled. In practice, the scenarios are represented using many different notations, for example, the *Unified Modeling Language* (UML) use case diagrams [20], message sequence charts, state machines, regular expressions and even using natural language expressions.

In the later sections, we will see details of our scenario-based testing approach.

3 Formal Software Development by Refinement

In this section, we present the formal refinement-based stepwise development approach. The main idea in this approach is to start the development from an abstract formal specification, which in a stepwise manner is transformed into an executable program. The correctness preserving steps of the formal development are called *refinements*. A formal specification, which is obtained from an informal system description, uses mathematical notations to describe the properties which a system must have. The stepwise development process is depicted in Figure 4. Since the refinement process preserves the system's correctness, the software developed by refinement is *correct by construction*.

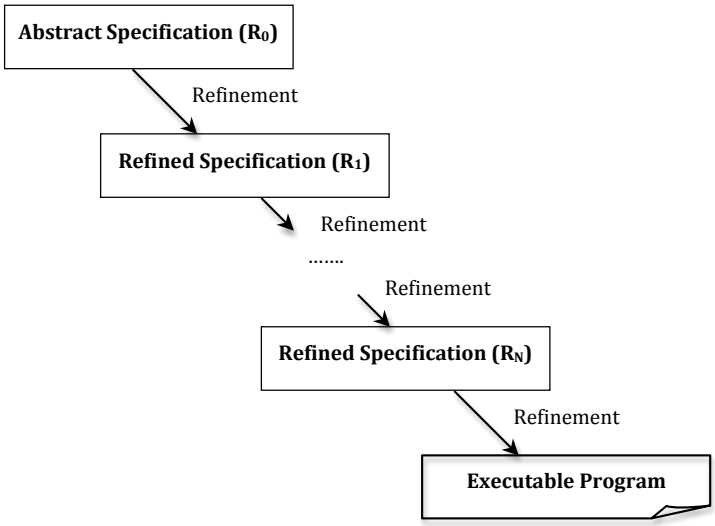


Figure 4: The Refinement Process

The idea of stepwise development was first proposed by Dijkstra [33] and Wirth [92] as an approach to develop correct programs. Later, Back [15] proposed the mathematical foundation of the refinement process, which was further developed into the refinement calculus framework by Back and Von Wright [17]. The refinement based development is formalized using the *weakest pre-condition* semantics [35].

Let S be a statement and P a post-condition predicate, i.e., a set of states which can be reached after executing S, then $wp(S, P)$ represents weakest pre-condition that guarantees establishing P after executing S. Now suppose that S is refined by T, then the refinement relation ($\sqsubseteq$) between S and T can be expressed [17] by the weakest pre-conditions as follows:

$$S \sqsubseteq T \text{ iff for all } P : wp(S, P) \Rightarrow wp(T, P)$$

where $\Rightarrow$ stands for implication between predicates. Informally speaking, a refined specification is said to refine an abstract specification if any post-condition, which abstract specification can establish, is also established by the refined specification. Alternatively, the refinement relation can also be expressed using *before-after* predicates relating initial and final states of statements. The expressions using before-after predicate will be presented in the next section.

The refinement process reduces non-determinism of the abstract specification and makes it more implementable. The transitivity of the refinement relation guarantees that each intermediate refined specification from R_1 to R_N as well as the *Executable* program are correct refinements of initial *Abstract Specification* (R_0) [17].

$$R_0 \sqsubseteq R_1 \sqsubseteq .... \sqsubseteq R_N \sqsubseteq Executable\ Program$$

The refinement can be categorized [17] into two forms: *data* and *algorithmic* refinement. The data refinement replaces an abstract data structure by the one which is more concrete and closer to the implementation, while preserving the global system behavior. On the other hand, algorithmic refinement introduces more concrete programming language structures to work on the data while leaving the structure of the data unchanged.

3.1 The Event-B Method

The Event-B [11, 10] is a recent extension of the classical B-method [9] formalism, which is a state-based formalism for specifying the system behavior as a special kind of a state transition system. The B-method has been successfully used in the industrial development of several complex real-life applications [18]. The Event-B has incorporated the Action System formalism [16] into the B-method, thus enabling reasoning about event-based (reactive) systems. The language of the B-method and Event-B is based on set theory and predicate calculus. The Event-B comes with a good tool support called the Rodin platform [7].

As an example of an Event-B model, consider the following model (also known as machine) M with a context C. A context is considered as the static part of the Event-B specifications. It contains constants, sets and properties (axioms) related to these. On the other hand, an Event-B machine describes the dynamic part of the specification in the form of events (state transitions). The context has the following general form.

CONTEXT C
SETS *sets*
CONSTANTS *constants*
AXIOMS *axioms*
END

A context is uniquely defined by its name in the **CONTEXT** clause. The **CONSTANTS** and **SETS** clauses define constants and sets respectively. The **AXIOMS** clause describes the properties of constants and sets in terms of set-theoretic expressions.
An Event-B machine has the following general form.

MACHINE M
SEES C
VARIABLES v
INVARIANT I
EVENTS
$INITIALISATION \quad = \quad \ldots$
$E_1 \quad = \quad \ldots$
$\ldots$
$E_N \quad = \quad \ldots$
END

A machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLES** clause defines state variables, which are then initialized in the **INITIALISATION** event. The variables are strongly typed by constraining predicates of the machine invariant I given in the **INVARIANT** clause. In addition, the invariant can define other essential system properties that should be preserved during system execution. The operations of event-based systems are atomic and are defined in the **EVENT** clause. An event is defined in one of two possible ways

$$E \quad = \quad \textbf{WHEN} \ g \ \textbf{THEN} \ S \ \textbf{END}$$

$$E \quad = \quad \textbf{ANY} \ i \ \textbf{WHERE} \ G(i) \ \textbf{THEN} \ S \ \textbf{END}$$

where g is a predicate over the state variables v, and the body S is an Event-B statement specifying how the variables v are affected by execution of the event. The second form, with the **ANY** construct, represents a parameterized event where i is the parameter (or a local variable) and *G(i)* restricts i. The event guard (e.g., g or *G(i)*) defines the condition under which execution of the event is enabled.

The occurrence of events represents the observable behavior of the system. The condition under which the action can be executed is defined by the guards. An event is known to be *enabled* when the guards evaluate to *true*. An event execution is supposed to take no time and no two events can occur simultaneously. When all events are disabled, i.e. their guards evaluate to false, the discrete system stops. When some events are enabled, one of them is chosen non-deterministically and its action modifies the state. Then previous step is repeated to see if any events are enabled for execution.

This can be summarized in the following.

```
Initialisation;
while (some events have true guards)
{
   Choose one such event;
   Modify the state accordingly;
}
```

An action can be either a deterministic assignment to the variables of the system or a non-deterministic assignment from a given set. The semantics of actions are defined by their before-after (BA) predicates and given in the following. A before-after (BA) predicate is a relation between *before* and *after* values of the event variables.

Action	**Before-after (BA) predicate**	**Explanation**
$x := F(x, y)$	$x' = F(x, y) \wedge y' = y$	*standard assignment*
$x :\in Set$	$\exists t.\ (t \in Set \wedge x' = t) \wedge y' = y$	*non-deterministic assignment from set*
$x :\mid P(x, y, x')$	$\exists t.\ (P(x, y, t) \wedge x' = t) \wedge y' = y$	*non-deterministic assignment by given post-condition*

where x and y are disjoint lists of state variables, and x', y' represent their values in the after state.

The $F(x, y)$ represents here a mathematical function that defines a new value for x (denoted by x') deterministically. The second part of the BA predicate requires all the remaining variables (y) should not change as a result of the assignment. The Set represents any defined set while $P(x, y, x')$ is a post-condition relating initial values of x and y to the final value x'. The $:\in$ and $:\mid$ represent non-deterministic assignment operators operating on sets and predicates respectively.

3.1.1 Proof obligations for specifications in Event-B

In order to check consistency of an Event-B machine, a number of pre-defined conditions (called *proof obligations*) should be proven true (i.e discharged) for each event [48, 68]. In recent practice, these proof obligations are generated and proved using the provided automated tool support. For each event in a machine, two types of properties are needed to be verified: the *event*

feasibility property and the *invariant preservation* property. The *event feasibility* states that it should be possible to execute an event from any state when both the machine invariant and the event guards hold. In other words, it can produce at least one after state that satisfies the before-after predicate, i.e.,

$$I(v) \wedge G_e(v) \quad \Rightarrow \quad \exists v'.\ BA_e(v, v') \tag{1}$$

where $I(v)$ is the invariant of the system while $G_e(v)$ is the guard of event e operating on variable(s) v. Similarly BA_e correspond to the before-after predicate of event e.

The *invariant preservation* property states that the invariant should always be maintained:

$$I(v) \wedge G_e(v) \wedge BA_e(v, v') \quad \Rightarrow \quad I(v') \tag{2}$$

The initialisation is treated as any other event of the system. The only difference is that it does not have initial state. Therefore, for the initialisation event, the *event feasibility*(1) and *invariant preservation*(2) properties become the following.

$$\exists v'.\ BA_{Init}(v') \tag{3}$$

$$BA_{Init}(v') \quad \Rightarrow \quad I(v') \tag{4}$$

In order to prove consistency of an Event-B machine, it is sufficient to prove the above stated proof obligations for all the events.

3.1.2 Refinement in Event-B

In Event-B, the systems are refined in a stepwise manner. Refinement process reduces non-determinism of an abstract specification by introducing concrete data structures and other implementation decisions. In other words, an abstract specification is gradually transformed (refined) into a less abstract one by introducing implementation details while preserving its correctness. A result of a refinement step is an independent model. Let us assume that the refinement machine M' is a result of refinement of the abstract machine M:

MACHINE M'
REFINES M
SEES C
VARIABLES w
VARIANT V
INVARIANT I'
EVENTS
$INITIALISATION \;=\; \ldots$
$E_1 \;=\; \ldots$
$\ldots$
$E_N \;=\; \ldots$
END

The variables in M' are denoted by w. The invariant I' of machine M' defines the invariant properties of the refined model. However, I' now also defines a connection between the newly introduced variables and the abstract variables that they replace. For a valid refinement step, every possible execution of the refined machine must correspond (via I') to some execution of the abstract machine [48]. In order to establish this we need two proof obligations, i.e., *feasibility of refined events* and *correctness* with respect to the abstract events. In order to show feasibility, we need to prove the following proof obligation.

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \;\; \Rightarrow \exists w'.\; BA'_e(w,w') \tag{5}$$

Here $G'_e(w)$ is the guard of the refined event and $BA'_e(w,w')$ is its before-after predicate. In order to show *correctness* of a refinement, we need to prove that the guards of the refined event are strengthened and there is a correspondence between the abstract and refined post conditions. These proof obligations are as follows:

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \;\; \Rightarrow \;\; G_e(v) \tag{6}$$

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \wedge BA'_e(w,w') \;\; \Rightarrow \exists v'.\; (BA_e(v,v') \wedge I'(v',w')) \tag{7}$$

The refinement can also introduce new events. The proof obligations for new events are similar to the above, however, we have to show that new events are refinements of implicit empty (*skip*) events of the abstract machine.

Finally, there is another proof obligation associated with the refined event that the new events can not take control forever and, hence, they have to terminate eventually when executed in isolation.

$$I(v) \wedge I'(v,w) \wedge G'_e(w) \wedge BA'_e(w,w') \;\; \Rightarrow V(w) \in \mathtt{NAT} \wedge V(w) > V(w') \tag{8}$$

Here V is a natural number expression that should be given in the special **VARIANT** clause of the refined machine. Proof obligation 8 requires that V decreases as a result of event execution, thus proving its eventual termination.

3.2 UML-B

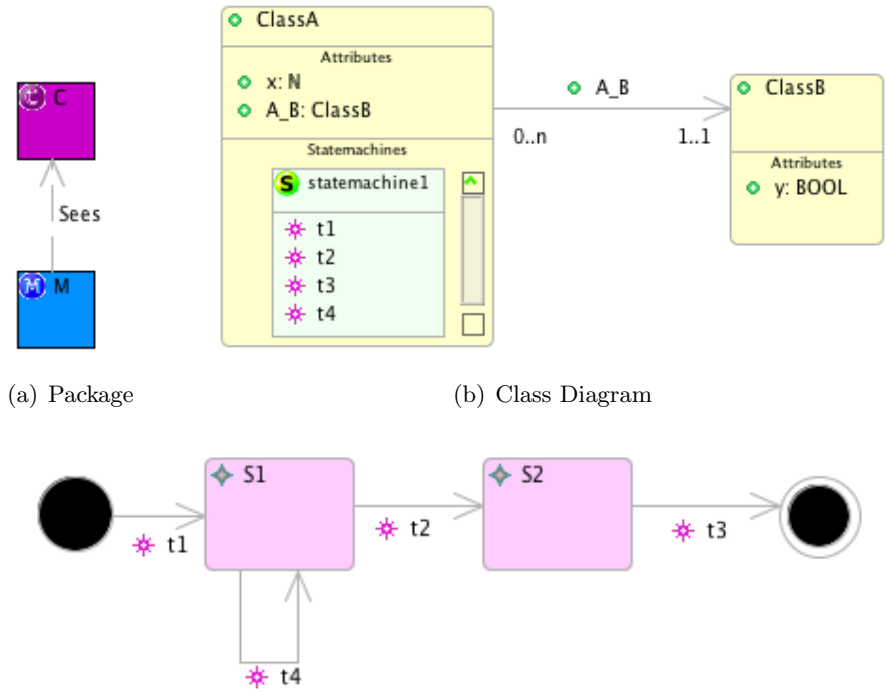
UML-B [79] is a new graphical language which combines certain UML features with Event-B. UML-B is similar to UML but has its own meta model. The main advantages of using UML-B is that it provides UML-like front-end to Event-B which might seem familiar to the majority of the developers. Moreover, it provides additional structuring of Event-B models in the form of UML classes and state-machines.

There are four kind of diagrams supported by UML-B, namely, context diagrams, package diagrams, class diagrams and state-machine diagrams. The package diagram is a top-level diagram which shows the structure and the relationship of components (contexts and machines). The *context diagram* describes static data such as constants and structured types. Logically, a UML-B context is similar to an Event-B context. The UML-B machine may contain *class diagram(s)* and *state-machine diagram(s)*. The *class diagram* may contain class *attributes* (variables), *associations* between classes, *methods* (events) and class-level *state-machines*. An attribute defines a data value of an instance of a class, whereas an association is a special kind of an attribute that defines a relationship between two classes.

The events, in object oriented terms, resemble methods of a class that modify the attributes of the class. A state-machine models the behavior of a class using transitions and discrete states. In addition to class-level state-machines, a UML-B machine may also contain one or more sub-state-machines. Moreover, a system *invariant* needs to be be defined in order to obtain complete formal specifications. The invariants and all other predicates in UML-B follow the same set-theoretic language as used by Event-B. However, in order to reflect some object oriented features, the Micro B (μB) [79, section 3.1] notation is used. For instance, object-oriented style *dot notation* is used to show class ownership of entities, such as attributes and associations.

UML-B comes with a good tool support in the form of a plug-in [80] for the Rodin platform [7]. The Figure 1.5(a) shows a package with a machine M and a context C. Examples of class diagram and state-machine are depicted in Figures 1.5(b) and 1.5(c) respectively.

The models in UML-B are automatically translated to the corresponding Event-B constructs. In Event-B everything is modeled in terms of contexts and machines. An UML-B context is translated as an Event-B context, whereas the class instance attributes from a UML-B machine become



(a) Package

(b) Class Diagram

(c) State-machine Diagram

Figure 5: Various UML-B Diagrams

variables in Event-B. The class associations are transformed into functions. Events and transitions in classes and state machines become events in the generated Event-B machine. The generated Event-B specifications are then proved using theorem provers.

Like Event-B, UML-B also follows stepwise refinement approach. An abstract machine in UML-B is refined further to include more details about structure and behavior of the system. A refinement may involve refinement of both classes and state-machines. A state-machine is generally refined to include sub-state machine(s). The modeling and refinement in UML-B are described in detail in [78].

4 Contributions of the Thesis

4.1 Scenario-based Testing and Formal Development

We combine the scenario-based testing, introduced briefly in Section 2.4, with formal system development. This approach can be divided into two parallel development processes, one for the formal development of models and second for the development of the testing scenarios.

The formal Event-B models are developed in a stepwise manner, starting with an abstract model and then refining it gradually in correctness-preserving refinement steps. The development of scenarios also follows the same stepwise development approach. A scenario is first specified at an abstract level. The abstract model, which is present at the same level of abstraction, must *conform* to this abstract scenario, meaning that the abstract scenario represents a valid behavior in the model.

In the formal development, the abstract scenario is refined along the refinement chain of the system models until a sufficiently detailed scenario is obtained. The overall process is presented in Figure 6. The scenarios are represented as Communicating Sequential Process (CSP) [76] expressions. The conformance relation between the model and the scenarios is checked using the ProB model-checker and animator [62].

The formal models are refined manually based on the guidelines, provided in our work (i.e., Publication II). However, the testing scenarios are refined automatically once the abstract scenario is provided by the user. This automatic refinement process is merely a syntactic transformation from the abstract scenario.

Once a sufficiently refined model is obtained, the implementation code can be generated from it. However, due to the abstraction gap between formal models and the executable implementations, automatic generation of implementation code is not always possible. Instead, an implementation is often hand-coded, while consulting the formal models. In our presented approach, we also propose a methodology for generating an implementation template, from the sufficiently refined model, that can be further developed to construct the complete implementation. The implementation template is generated considering the final implementation to be in the Java programming language. However, the template generation approach can also be adapted for other programming languages such as Python and C#.

In order to make sufficiently refined scenarios executable, they need to be concretized into executable test cases. In our approach, we *concretize* the scenarios into unit tests [22]. Since we consider Java for our implementation template, it was natural to choose the Java Unit testing framework (JUnit) [5] for this purpose.

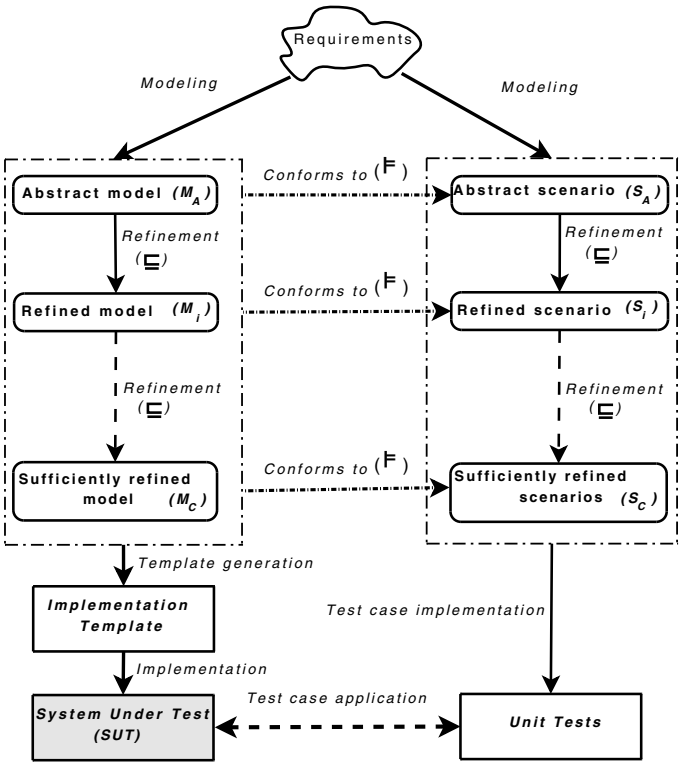


Figure 6: Our Scenario-based Testing Approach

In addition to the above, we also propose a methodology for associating informal requirements to the model elements and testing scenarios. These requirements are presented in textual format and are propagated from the model and the scenarios to the implementation template and the unit test cases respectively. The main purpose of this approach is to be able to trace scenarios to different parts of the specification and, in the end, to the generated test cases.

4.1.1 Related Work

The scenarios are used during various phases of the software development. In the field of specification and behavior modeling, *message sequence charts* (MSCs) [29] and UML sequence diagrams [71] are one of the most popular techniques to model scenarios. These are quite helpful in designing concurrent and distributed systems since these show interrelationships between different components, object instances or processes - and between them and the environment. MSCs have been extended or modified for use in other modeling and structuring approaches and one of the notable works in this direction is *live sequence charts* (LSCs) [50]. LSCs describe scenarios in the terms of liveness i.e., the things that must occur, by using two types of

charts called universal and existential charts. LSCs are executed into a LSC interpreter called "Play Engine".

Lee et al.[61] used Petri nets for the analysis and integration of use case scenarios. This approach defines Constraint-based Modular Petri Nets (CMPNs) and presents a procedure to create CMPNs from scenarios. The CMPNs are then used for checking consistency and completeness of the system. This approach is used for validation purpose and not for test generation.

Hall [47] in his work on scenarios, describes a method and tool for building, managing and using *large scale scenarios* (LSS). In this approach, the scenario groups are represented by parameterized scenario classes. The scenarios are animated visually and validation of the scenarios is done by "constraint checking". In this work, the scenarios are used for requirement engineering purpose.

The use of scenarios in the field of software testing is as old as the testing itself. The only difference between the traditional-styled manual testing and the automated scenario-based testing is that in the former, the testing scenarios are in the mind of test engineers while in the latter, these are documented and connected with other modeling structures. The work related to scenario usage in automated testing is presented in the following.

In [83], a Scenario-based Object Oriented Testing Framework (SOOTF) is proposed. The scenarios are described as tree structures where nodes of the tree represent functions of the UML classes under test. The scenarios are transformed into executable test cases and stored in the database to be used again for regression testing. A similar approach to SOOTF is proposed by Hsia et al. [54], where scenarios are represented as trees. This approach is based on regular grammars and state machines.

In [77], a method for SCENario-based validation and Testing (SCENT) of software is presented. In this approach, the scenarios are described as natural language expressions which are later formalized in the form of state charts. The tests are derived from the state charts by using path traversal techniques.

TOBIAS [60] is a combinatorial testing tool that also combines scenarios as sequence of method calls. TOBIAS is used for conformance testing of VDM [57] and JML [59] specifications. The test case generation uses assertions and pre-conditions from the specifications to filter the test case generation process.

The jSynoPSys tool [31] performs scenario-based testing using symbolic animation of the B machines. The scenarios are represented using scenario-descriptive language. This work is based on CLPS-BZ [25] constraint solver. Although the B-method is used for the specifications, however, this work does not describe how the refinement of scenarios or specifications is taken into account.

4.2 Modeling with UML-B for Model-based Testing

In this work, we use UML [71] and UML-B [79] models for model-based testing. The purpose of this work is to improve the quality of the models used in the MBT process by incorporating formal verification. Figure 7 depicts the overall process used in our approach. The UML models are used for modeling architectural and static aspects of the SUT, whereas UML-B is used to formally verify the behavioral aspects of the system.

Once these models are constructed, they are transformed into input for Conformiq's Qtronic [55], which is a tool for model-driven test case design. Qtronic generates test cases from the specifications of the SUT, which are represented using *Qtronic Modeling Language* (QML). QML uses a Java-like action language and state-machine models to represent the behavioral part of the SUT. At the later phase, Qtronic generates the test cases and applies them on the SUT in online mode.

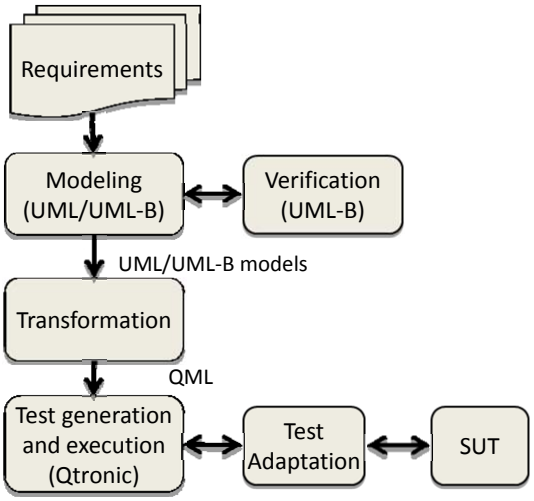


Figure 7: Our Scenario-based Testing Approach

Although this methodology has been proposed for using UML and UML-B for modeling, and Qtronic for test generation, it can be generalized to be used with other modeling and test generation techniques.

4.2.1 Related Work

Microsoft's SpecExplorer [88] is a tool for model-based specification and conformance testing. The system's abstract behavior is encoded in machine-executable form, which is also referred to as "model-program". A model-program defines state variables and update rules of an abstract state ma-

chine. The tool uses explicit-state model checking technique to explore the states of the state machine. The results of the exploration can be visualized graphically and test cases are generated from these results.

Object-Z [38], which is an object-oriented extension of the formal specification language Z [81], has been used for test generation in [28, 66], however, these approaches do not use graphical models and lack tool-support for industrial-scale test generation.

In MATERA [8] approach, UML models (requirement, domain, architecture and test configuration) are used in the model-based testing process. The models are validated using OCL [70] constrains. The validated models are then transformed into the input for Conformiq's Qtronic [55] tool for automated test generation. This work supports requirement traceability from the failed test cases to the UML model elements.

LEIRIOS Test generator (LTG) [56], now called SmartTesting, uses B formal models for model-based testing. The B models, which encode the system's behavior, are validated using LTG symbolic animator. The abstract tests are generated by covering all symbolic execution paths for all operations with input parameters' boundary values. The generated test cases are later translated into executable test scripts.

In [42], a conformance testing and automatic test case generation scheme for UML Statecharts (UMLSCs) is presented. The authors propose a formal conformance-testing relation for input-enabled transition systems with transitions labeled by input/output-pairs (IOLTSs). The test case generation algorithm is a mix of process algebra and a simplified version of lambda calculus. This work, like other state chart based approaches, does not consider other aspects, e.g., architectural, data etc., of the SUT.

4.3 Comparative Study of Modeling Subjects in Model-based Testing

As it has already been discussed in Section 2.3, in model-based testing, the modeling subject can be the SUT itself or the environment where the SUT operates. In this work, we analytically compare two model-based testing approaches using different modeling perspectives. Moreover, we describe how these two types of models are constructed and used in the MBT process. We try to find out how the difference in the viewpoints affect the modeling process and the actual testing. The two perspectives are studied in the context of two case studies, both from the telecommunication domain. In the first case study, MBT is applied to test functional features of Mobile Switching Server (MSS). The second case study employs model-based testing for testing a smartphone application by using its Graphical User Interface (GUI).

4.3.1 Related Work

Model-based testing has been in the industry and academia for more than a decade and today several commercial and academic tools exist. However, we have not found any related work comparing differences between using the SUT model and the environment model for model-based testing. There have been few comparisons in the literature between different model-based testing tools and techniques, however, these do not directly relate to our work presented in this thesis. These are presented in the following.

In [51], a list of model-based testing tools is presented. This list contains both the academic and the commercial tools. Each tool in the list is briefly described discussing the input model and test generation technology it uses.

Dias-Neto et al.[32] conducted a literature survey and defined many dimensions for characterizing model-based testing approaches. The authors also developed a tool support for selection of model-based testing techniques according to the needs and the project nature.

In [44], the authors compared the efforts required by different modeling scenarios in model-based testing. The comparison is based on a test manager's point of view focusing on testing activities and organization aspects. This work presents six different scenarios focusing on the origin of the model (e.g., development model, separate model, reverse-engineered model etc.), used for the model-based testing, with advantages and disadvantages these bring in the process.

4.4 Overview of the Papers

The contribution of the thesis is described in the context of the individual research papers, which are presented in Part II of this thesis. Each paper is summarized and the author's contributions are discussed.

4.4.1 Paper I: Synthesis of Scenario Based Test Cases from B Models

Paper I was a first attempt to explore the possibility of test refinement while the system specifications are developed using formal refinement steps. The system specifications are expressed using the Event-B formalism, whereas the provided testing scenarios are expressed in a simple textual form, listing the operations (events) of the system in some sequence. The abstract scenario is described at the level of an abstract specification. When this specification passes through a succession of refinements, we derive the scenario-based test cases for each refinement and show that all these test cases are equivalent to the test cases of the original specification. The refinement of test cases is performed by using a proposed algorithm, which searches for the new operation instances added during the refinement process. However,

the presented algorithm has proved to be exponential in nature thus limiting its application to larger specifications.
Software and tools' setup: As a case study example for this work, the leader-election problem was modeled in Event-B with three levels of refinements. The Event-B specifications were proved correct using AtelierB [1] provers before the proposed algorithm was applied.
Author's contribution: The basic idea presented in this paper was coined by Prof. Johan Lilius and Dr. Manoranjan Satpathy and was jointly developed by all the contributors listed in the paper. The author took part in the discussions and development of the algorithm.

4.4.2 Paper II: Model-based Testing Using Scenarios and Event-B Refinements

In this paper, we improved our scenario-based testing approach introduced earlier in Paper I. The basic idea remains the same - we refine the user-provided testing scenarios automatically at each step together with the corresponding system specifications. The specification language, i.e., Event-B, remains the same while the testing scenarios are represented as Communicating Sequential Process (CSP) expressions. Representing scenarios as CSP expressions gives us better structure and the associated tool support. In this work, we also described basic refinement rules, allowing us to do the development in a controlled way and to transform our testing scenarios according to those rules. At each refinement step, the testing scenarios are checked for conformance against specifications, using the ProB model-checker and animator. The presented approach does not involve any exponential algorithm thus making it more applicable in practice. The approach is exemplified by a case study example of the development of a fault tolerant agent system.
Software and tools' setup: For this work, we used the case study of a mobile-agent system. The Event-B specifications were proved using provers within the Rodin platform. Moreover, as described earlier, the CSP expressions were checked for conformance against specifications by using the ProB model-checker and animator.
Author's contribution: The author developed this idea and wrote the paper under the supervision and guidance of Prof. Johan Lilius and Dr. Linas Laibinis.

4.4.3 Paper III: Requirement-driven Scenario-based Testing Using Formal Stepwise Development

Paper III is an extension of Paper II. In Paper II, the testing scenarios were presented on abstract level, while in Paper III the testing scenarios are transformed into executable test cases. In order to make test cases executable,

more information on the inputs and outputs of the system is needed. Therefore, the procedure for system development by refinement is appended with a new concept of *Input-Output Unit* (IOUnit). The sufficiently refined specifications are translated into concrete implementations. In this work, we also proposed an approach to generate Java language implementation templates from the given Event-B models. Moreover, the abstract testing scenarios are translated into executable JUnit test cases.

Additionally, we also proposed a mapping functionality, to map informal requirements to the formal model and to the testing scenarios at different refinement steps. This mapping of informal requirements is extended down to the concrete test cases. When a test case fails, the unfulfilled requirements can be back-traced into the model. In particular, the paper proposed the following:

- inclusion of requirements in the formal specification process and propagation of requirements to tests;
- a method for identification of abstract test cases from formal scenario specifications;
- a method for generating Java templates of the system from sufficiently refined Event-B specifications;
- a method for generating JUnit tests from abstract test cases in CSP.

Software and tools' setup: This work was supported by a small case study example of a hotel-booking system. The Event-B specifications, with several levels of refinements, were developed and proved using the Rodin platform and its supported plug-ins. Moreover, we used JUnit and TestNG testing frameworks for executable test cases. The code-coverage analysis was performed by using EclEmma [3] tool. In order to represent and map informal requirements to formal specifications, we used the requirement management plug-in, named *ReqsManagement*, of the Rodin platform.
Author's contribution: The author further extended the idea of scenario-based testing and wrote this paper under the supervision and guidance of Prof. Johan Lilius and the listed co-authors.

4.4.4 Paper IV: Using UML Models and Formal Verification in Model-Based Testing

In this paper, we present a model-based testing approach where we integrate UML, UML-B and the Qtronic test generator. This paper, like previous ones, uses a refinement-based approach where details are added in the model in a stepwise manner. However, in this paper, we use UML-B

to model the behavioral part of the system, while the architectural models of the system are specified in UML. The UML-B models, which mainly consists of class and state machine diagrams, are automatically translated into Event-B specifications that can be proved using theorem provers. Once the formal models are proved, the UML-B models are transformed into the QML notation used by the Qtronic test generator. This approach combines UML modeling with formal verification in order to improve the quality of the models used for automated test derivation. The proposed methodology is illustrated by using excerpts from a telecommunication case study.
Software and tools' setup: The case study used in this work was the *Location Update* feature of a Mobile Switching Server (MSS). For UML modeling, we used the MagicDraw [6] tool, while for developing UML-B models, we used the UML-B plug-in of the Rodin platform. The UML-B models were developed using refinement-based approach and all of the models were proved using available provers within the Rodin platform. For test generation, Qtronic tool was used.
Author's contribution: The author developed this idea and wrote the paper under the supervision and guidance of Dr. Dragoş Truşcan and Prof. Johan Lilius.

4.4.5 Paper V: Model-Based Testing using System vs. Test Models -What is the difference?

In Paper V, we discuss the differences between using *"system models"* and *"test models"* with respect to the model-based testing process. System models describe the internal behavior of the system under test, whereas test models specify the system behavior from the point of view of the user or of the environment. We analyze how these two types of models are obtained and used throughout the model-based testing process and how they are related to each other. The discussion is based on authors' earlier experiences as well as on two case study examples from the telecommunication domain. The idea of this paper emerged as authors were comparing different types of models and notions used for model-based testing, in particular, when focusing on different perspective of the SUT behavior.
Software and tools' setup: This research work was theoretical in nature and hence, did not require any software or tool setup. However, the original case study examples, which we compared in this paper, did involve software and tool setups.
Author's contribution: Paper V is a joint work between Åbo Akademi University and Tampere University of Technology. The study was done on two model-based testing case studies, one from each university, and the findings were analyzed and compared with each other. The author was responsible for collecting and writing most of the analytical details of two case studies

under guidance of Dr. Dragoş Truşcan and Adj. Prof. Mika Katara. The two case studies were compared for the purpose of the presented research, however, originally the two case studies were developed independently of each other by the listed co-authors in their respective universities.

4.5 Mapping over Taxonomy of Model-Based Testing

In the following, we map our work, presented in this thesis, over the taxonomy of the MBT already discusses in Section 2.3. We discuss the categorization from Paper I to Paper IV. The Paper V is not included in this mapping as it is does not propose any model-based testing methodology.

Subject of the model

The model we use are the *behavioral models* of the SUT.

Model redundancy level

We use Event-B formal models for our approach. These models are usually constructed for the purpose of development of the system. In our case, these are *shared* for both development and for testing. Our decision to use shared models was motivated by the fact that Event-B specifications were quite detailed, precise and fully-proved before being used for testing. Moreover, in scenario-based testing, the user-provided scenarios represent another parallel layer of models which are checked for conformance with the Event-B models, hence, limiting chances of ambiguities. However, constructing separate models for development and testing are likely to find more ambiguities of the specifications.

Model Characteristics

The models we have used are *deterministic*, *un-timed* and *discrete* in nature.

Modeling Paradigm

We use Event-B formal models in all of our approaches. The Event-B is based on the *Pre-Post* semantics. However, the Event-B models can also be thought of as *transitions-based models*, e.g., state-machines or labelled transition systems. For instance, it can be observed that the UML-B uses transition-based notation (state diagrams), however, it translates to Event-B in the background.

Test Selection Criteria

We used *requirement-based* selection criteria as the test scenarios represent the user requirements which are to be tested. The test scenarios are in fact test cases which are specified abstractly hence they can be categorized as following the *Test case specification* criteria.

Test Generation Technology

We use a mix of various technologies in our work. In Paper I, we used a *search-based* algorithm to find refinement of the test cases. While in Paper II and III, we rather *derive* our test cases from abstract test scenarios and formal models. We use *model-checking* or more precisely *model-animation* to check the validity of the abstract test cases. The *theorem provers* are used to prove the formal Event-B models rather than to generate test cases. The abstract test cases are transformed into executable JUnit tests using syntactic transformation.

In Paper IV, we transform the models from UML to Qtronic. The Qtronic test generator uses *random generation* and *graph-search algorithms* as test generation technology [85, page 377].

Test Execution Mode

At the moment, the scenario-based testing methodology, presented in Paper I, II and III, considers offline execution of the test cases since the test cases will be generated and stored in the form of JUnit test cases. However, in Paper IV, the tests generated by Qtronic can be executed in either Online or Offline mode depending on the test setup.

5 Discussion

5.1 Testing vs. Formal Verification

Testing is an incomplete process as exhaustive testing is not always practical. On the other hand, formal verification had mostly been an academic or research topic. In the past, the testing and formal verification had often been considered as rivals [26, Section 1.1]. However, in recent years these have appeared as complementary techniques [24].

Formal methods no doubt provide rigorous mathematical techniques for proving correctness, but formal verification alone does not guarantee the correctness of the real system as described by Hall in Seven Myth's of Formal Methods [46, Myth 1]. The reason behind this is the fact that formal specifications of the system might be incorrect or incomplete, i.e. they do not include everything that it takes for a system to be correct. In addition

to that, due to immense complexity of software, the specifications represent an abstract view of the actual software. Since the abstract model hides some of the implementation details, it may also hide the associated errors. In this context, there remains a need for testing even when the formal techniques are used for software development. That was the reason why in the famous Ten Commandments of Formal Methods [27], the Commandment IX states, *"Thou shalt test, test and test again"*.

The formal verification finds inconsistencies and errors at a certain level of abstraction. On the other hand, in testing, the system is executed in its real environment with conditions as close as possible to its intended use, thus testing is good at finding some errors at all levels of abstraction including hardware [84]. Figure 8 depicts the relation between software testing and formal verification in context of finding errors at different abstraction levels.

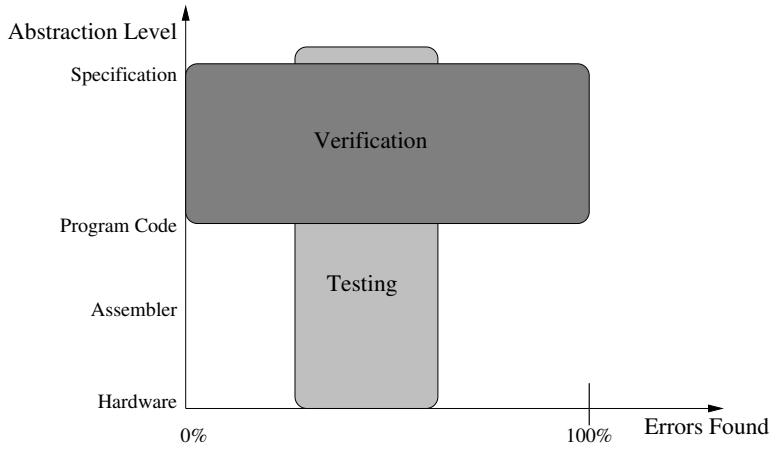


Figure 8: Testing vs. Formal Verification from [84]

5.2 Model-Based Testing vs. Formal Verification

The model-based testing is an advanced kind of testing, therefore, our arguments related to the importance of testing from previous section are still valid here. The model-based testing advocates the usage of models for test generation. The model-based testing tools [85], which are employed in industry, use formal or semi-formal models for this purpose. Hence, model-based testing can be considered as a first step towards introduction of formal methods in those sects of industry where no formalism was used before. With the recent increased adaption of model-based testing in industry, we can hope that in future, software professionals will start using more formal or semi-formal techniques.

It is commonly believed that formal methods are applicable to system of a limited size and full automation of formal methods in not always possible. Model-based testing can still be used even in the presence of a such fully automated formal development method. In this case, the model-based testing can be used as a tool for validation of specifications and requirements and can be applied on the models before verification is started. It has been revealed by the studies that most of the faults exposed by MBT are specification and requirement errors rather than implementation errors [21].

6 Summary and Conclusions

Most of the work presented in this thesis is on combining model-based testing with the formal stepwise development approach. Moreover, we also look at different modeling perspectives used for model-based testing. The described methods and techniques have been presented at international conferences and published in a refereed journal. A selection of representative papers has been collated and attached to this thesis.

The presented work can clearly be divided into three parts. The first part deals with the scenario-based testing where the test cases are generated from the user-provided testing scenarios, while behavior of the system is specified in Event-B formalism using stepwise development approach. The testing scenarios are presented by the user at the abstract level, corresponding to the abstract Event-B specification. Once an abstract scenario is constructed, it is automatically refined following the refinement step used to refine Event-B specification. In order to facilitate the automatic refinement, we have described basic refinement rules, allowing us to do the development in a controlled way and transform testing scenarios according to those rules. Additionally, we also proposed an approach to generate Java language implementation templates from Event-B models. The abstract testing scenarios can then be used to generate executable JUnit test cases. Optionally, the user can map informal requirements to the formal model and testing scenarios at different refinement steps. This mapping of informal requirements is extended to concrete test cases so that upon test case failure, these unfulfilled requirements can be back-traced into the model.

Our scenario-based testing approach brings the following benefits to the software development process.

- We use formal Event-B models where complexity of the system to be developed is handled by refinement-based stepwise development and the models are formally verified.
- In traditional model-based testing approaches, the whole process is based on the coverage criteria, such as transition coverage, state coverage or any other variation of these. Our scenario-based approach

can be seen as an attempt to focus the testing process, by explicitly identifying important behavior of the system that should be tested.

- We propose a way to include informal requirements in the formal specification process and propagation of requirements to the tests. The back-traceability of requirements, from the failed test cases to the model elements, is quite helpful in the debugging process.

- A method for generating Java templates of the system to be developed from sufficiently refined Event-B specifications, provides a starting point for the implementation, thus, saving overall development time.

- Automatic generation of JUnit tests based on user-provided scenarios, on the one hand, ensures that the important behaviors of the system are tested while, on the other hand, it saves overall time used for the testing.

In the second part of the thesis, we proposed another approach for combining model-based testing with the formal stepwise development. In this case, the UML and UML-B models are used to specify architectural and behavioral aspects of the system, respectively, and Conformiq's Qtronic is used as the test generation tool. The UML-B models, comprising mainly of state machines, are developed in a stepwise manner. These UML-B models are then automatically translated into Event-B specifications that can be proved using theorem provers. The basic idea to use UML-B state machines, instead of UML state machines, is to increase the quality of the models used for test derivation, by incorporating formal verification in the process.

The last part of the thesis deals with a comparative study of two modeling perspective used for model-based testing. In the first perspective, the models, termed as "system models", are developed from the perspective of the implementation, while in the second perspective, the models, termed as "test models", see the implementation as a black-box. The comparison is carried out using two case studies from the telecommunication domain.

Generally speaking, the model-based testing brings number of benefits, some of these are listed in the following.

- The overall time for the development process is shortened while forcing the testability into the product design.

- The construction of the behavioral models of the system-under-test may help in finding the specification and design bugs and resolving requirement ambiguities.

- Since the tests are generated automatically, different coverage criteria can be applied to increase testing thoroughness while there is no cost associated with the test suite maintenance.
- The changes in the design can also be handled with lesser effect, as just the model needs to be changed and test cases can be re-generated from the updated model.

Model-based testing has a good potential to enhance the quality of the complex software products. However, the model-based testing still needs to go a long way before it can become everyday practice in the software industry. The adaptability of model-based testing is hindered due to the following reasons.

- One of the main hinderances is the lack of modeling practice by testing engineers in the industry, in particular, practice of formal and semi-formal modeling, which is essential for using model-based testing.
- There are also limitations imposed by model-based testing tools and techniques, for instance, modeling restrictions in terms of abstraction handling and modeling language notations. Test generation techniques also suffer from limitations, e.g., state-space explosion problem and limitations on translation of abstract test cases to low-level executable test cases.
- At the organization level, adapting model-based testing often requires re-organization of the software design process. It also requires employment of skilled professionals or effective training of the existing team.

In conclusion, the model-based testing aims at improving the quality of the software with the overall gain in the productivity. The work presented in this thesis is a step towards this direction.

Bibliography

[1] AtelierB - User Manual. ClearSy System Engineering, Aix-en-Provence, France. `http://www.atelierb.eu/php/manuels-atelier-b-en.php` Accessed in August 2010.

[2] Deployment of Model-Based Technologies to Industrial Testing (D-MINT). ITEA2 project, online at `http://www.d-mint.org/` Accessed in August 2010.

[3] EclEmma - Java Code Coverage for Eclipse. `http://www.eclemma.org/` Accessed in August 2010.

[4] ITEA2 - Information Technology for European Advancement. `http://www.itea2.org/` Accessed in August 2010.

[5] Java Unit Testing - JUnit 4. `http://www.junit.org/` Accessed in August 2010.

[6] NoMagic MagicDraw. `http://www.magicdraw.com/` Accessed in August 2010.

[7] Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, online at `http://rodin.cs.ncl.ac.uk/` Accessed in August 2010.

[8] Fredrik Abbors, Andreas Backlund, and Dragos Truscan. MATERA - An Integrated Framework for Model-Based Testing. *IEEE International Conference on the Engineering of Computer-Based Systems*, pages 321–328, 2010.

[9] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[10] J.-R. Abrial. A System Development Process with Event-B and the Rodin Platform. In *International Conference on Formal Engineering Methods*, pages 1–3, 2007.

[11] J.-R. Abrial. *Modeling in Event-B: System and Software Design.* Cambridge University Press, 2010.

[12] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[13] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[14] James D. Arthur, Markus K. Gröner, Kelly J. Hayhurst, and C. Michael Holloway. Evaluating the Effectiveness of Independent Verification and Validation. *Computer*, 32(10):79–83, 1999.

[15] R. J. R. Back. *On Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, Finland, 1978.

[16] R. J. R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM.

[17] Ralph-Johan Back and Joakim von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In *REX ((Research and Education in Concurrent Systems) workshop: Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 42–66, Mook, The Netherlands, 1990. Springer-Verlag New York, Inc.

[18] Patrick Behm, Pierre Desforges, and Jean-Marc Meynadier. MÉTÉOR: An Industrial Success in Formal Development. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, page 26, London, UK, 1998. Springer-Verlag.

[19] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company Limited, 1990.

[20] Kurt Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] Mark Blackburn, Robert Busser, and Aaron Nauman. Why Model-based Test Automation is Different and what you should know to get started. In *In International Conference on Practical Software Quality and Testing*, 2004.

[22] IEEE Standards Board. IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987. In *IEEE Standards: Software Engineering, Volume Two: Process Standards*. The

Institute of Electrical and Electronics Engineers, Inc. Software Engineering Technical Committee of the IEEE Computer Society, 1987.

[23] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[24] K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, R. M. Hierons, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Working together: Formal Methods and Testing. *ACM Computing Surveys*, 2003.

[25] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. CLPS-B: A Constraint Solver to Animate a B Specification. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):143–157, 2004.

[26] J. Bowen, K. Bogdanov, J. Clark, M. Harman, R. Hierons, and P. Krause. FORTEST: Formal Methods and Testing. *Annual International Conference on Computer Software and Applications*, page 91, 2002.

[27] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods...Ten Years Later. *Computer Journal, issn: 0018-9162*, 39:40–48, 2006.

[28] David Carrington, Ian Maccoll, Jason Mcdonald, Leesa Murray, and Paul Strooper. From Object-Z Specifications to ClassBench Test Suites. *Journal on Software Testing, Verification and Reliability*, Vol. 10:111–137, 2000.

[29] Message Sequence Charts. International Telecommunications Union (ITU). Telecommunication Standardization Sector. 1996.

[30] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2008.

[31] Frédéric Dadeau and Régis Tissot. jSynoPSys – A Scenario-Based Testing Tool based on the Symbolic Animation of B Machines. *Electron. Notes Theor. Comput. Sci.*, 253(2):117–132, 2009.

[32] Arilo Claudio Dias-Neto and Guilherme Horta Travassos. Model-based Testing Approaches Selection for Software Projects. *Information and Software Technology*, 51(11):1487 – 1504, 2009. Third IEEE International Workshop on Automation of Software Test (AST 2008); Eighth International Conference on Quality Software (QSIC 2008).

[33] E. W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. *BIT Numerical Mathematics, issn: 0006-3835, Springer Netherlands*, 8:174–186, 1968.

[34] E. W. Dijkstra. On the Reliability of Mechanisms. *Notes On Structured Programming, EWD249*, 1970. available at `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF` Accessed in August 2010.

[35] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[36] Mark Dowson. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997.

[37] David A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, Inc., New York, NY, USA, 1991.

[38] Roger Duke and Gordon Rose. *Formal Object Oriented Specification Using Object-Z*. Palgrave macmillan, 2000.

[39] Sean Feinberg Elsje Scott, Alexander Zadirov and Ruwanga Jayakody. The Alignment of Software Testing Skills of IS Students with Industry Practices A South African Perspective. *Journal of Information Technology Education*, 3, 2004.

[40] E. Allen Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1995.

[41] P.G. Frankl and E.J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[42] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal Test-Case Generation for UML Statecharts. *IEEE International Conference on Engineering of Complex Computer Systems*, pages 75–84, 2004.

[43] Object Management Group. UML Testing Profile, v 1.0. formal/05-07-07.

[44] Baris Guldali, Michael Mlynarski, and Yavuz Sancar. Effort Comparison for Model-Based Testing Scenarios. *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 28–36, 2010.

[45] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language LUSTRE. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, 1991.

[46] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[47] Robert J. Hall. A Method and Tools for Large Scale Scenarios. *Automated Software Engg.*, 15(2):113–148, 2008.

[48] S. Hallerstede. The Event-B proof obligation generator. In L. Viosin, editor, *Specification of Basic Tools and Platform, RODIN Deliverable 3.3 (D10), RODIN-Project, IST-511599*, 2005.

[49] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[50] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[51] Alan Hartman. Model based test generation tools, available at: `http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf` Accessed in August 2010.

[52] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341, London, UK, 2002. Springer-Verlag.

[53] W.E. Howden. Systems Testing and Statistical Test Data Coverage. *Annual International Conference of Computer Software and Applications*, pages 500–504, 1997.

[54] Pei Hsia, Jayarajan Samuel, Jerry Gao, David Kung, Yasufumi Toyoshima, and Cris Chen. Formal Approach to Scenario Analysis. *IEEE Softw.*, 11(2):33–41, 1994.

[55] Conformiq Inc. Qtronic. `http://www.conformiq.com/qtronic.php` Accessed in August 2010.

[56] Eddie Jaffuel and Bruno Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. *B 2007: Formal Specification and Development in B*, pages 277–280, 2006.

[57] Cliff B. Jones. *Systematic Software Development Using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[58] Axel van Lamsweerde. Formal specification: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM.

[59] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[60] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering TOBIAS Combinatorial Test Suites. *Fundamental Approaches to Software Engineering*, pages 281–294, 2004.

[61] Woo Jin Lee, Sung Deok Cha, and Yong Rae Kwon. Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering. *IEEE Transaction on Software Engineering*, 24(12):1115–1130, 1998.

[62] M. Leuschel and M. Butler. ProB: A model checker for B. Proc. of FME 2003, Springer-Verlag LNCS 2805, pages 855-874., 2003.

[63] Bao-Lin Li, Zhi shu Li, Li Qing, and Yan-Hong Chen. Test Case Automate Generation from UML Sequence Diagram and OCL Expression. *International Conference on Computational Intelligence and Security*, pages 1048–1052, 2007.

[64] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[65] Matlab Simulink. `http://www.mathworks.com/` Accessed in August 2010.

[66] Jason Mcdonald, Leesa Murray, and Paul Strooper. Translating Object-Z Specifications to Object-Oriented Test Oracles. In *In Proc. Asia-Pacific Software Engineering Conference and Int. Computer Science Conference*, pages 414–423. IEEE Computer Society, 1998.

[67] Kwan Mei-Ko. Graphic Programming Using Odd or Even Points (Chinese Postman Problem). *Chinese Math*, 1:273–277, 1962.

[68] C. Métayer, J.-R. Abrial, and L. Voisin. Event-B language. In *RODIN Deliverable 3.2 (D7), RODIN-Project, IST-511599*, 2005.

[69] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, Inc., 1979.

[70] Object Constraint Language (OCL) Version 2.0, Object Management Group. `http://www.omg.org/technology/documents/formal/ocl.htm` Accessed in August 2010.

[71] Documentation of the Unified Modeling Language (UML). available from the object management group (omg). `http://www.omg.org` Accessed in August 2010.

[72] V. Papailiopoulou. Automatic Test Generation for LUSTRE/SCADE Programs. In *ASE 08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 517–520, Washington, DC, USA, 2008. IEEE Computer Society.

[73] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[74] Alexander Pretschner. Classical search strategies for test case generation with constraint logic programming. In *In Proc. Formal Approaches to Testing of Software*, pages 47–60. BRICS, 2001.

[75] Recall on 2010 Model-Year Prius and 2010 Lexus HS 250h Vehicles to Update ABS Software. `http://pressroom.toyota.com/pr/tms/toyota-2010-prius-abs-recall-153614.aspx` Accessed in August 2010.

[76] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998 amended 2005.

[77] Johannes Ryser and Martin Glinz. A Practical Approach to Validating and Testing Software Systems Using Scenarios. In *QWE 99, 3rd International Software Quality Week Europe*, 1999.

[78] Mar Yah Said, Michael Butler, and Colin Snook. Class and State Machine Refinement in UML-B. In *Integration of Model-based Formal Methods and Tools (workshop at iFM 2009)*, February 2009.

[79] Colin Snook and Michael Butler. UML-B: Formal Modeling and Design Aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[80] Colin Snook and Michael Butler. UML-B: A plug-in for the Event-B tool set. In *Abstract State Machines, B and Z, First International Conference ABZ 2008*, page 347, September 2008.

[81] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[82] Jan Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, 1996.

[83] W. T. Tsai, A. Saimi, L. Yu, and R. Paul. Scenario-based Object-Oriented Testing Framework. *International Conference on Quality Software*, page 410, 2003.

[84] Mark Utting. Position Paper: Model-based Testing. *Verified Software: Theories, Tools, Experiments. ETH Zürich, IFIP WG*, 2, 2005.

[85] Mark Utting and Bruno Legeard. *Practical Model-Based Testing.* Morgan Kaufmann Publishers, 2006.

[86] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-based Testing. Technical report, 2006. ISSN 1170-487X, The University of Waikato `http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf` Accessed August 2010.

[87] R. J. van Glabbeek. Notes on the methodology of CCS and CSP. In *ACP '95: Proceedings from the international workshop on Algebra of communicating processes*, pages 329–349. Elsevier Science Publishers B. V., 1997.

[88] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. *Formal Methods and Testing*, pages 39–76, 2008.

[89] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme. *Modeling Software with Finite State Machines.* Auerbach Publications, Boston, MA, USA, 2006.

[90] Anders Wesslén, Per Runeson, and Björn Regnell. Assessing the Sensitivity to Usage Profile Changes in Test Planning. *International Symposium on Software Reliability Engineering*, page 317, 2000.

[91] James A. Whittaker and Michael G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.

[92] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

Part II

Original Publications

Paper I

Synthesis of Scenario Based Test Cases from B Models

Manoranjan Satpathy, Qaisar A. Malik and Johan Lilius

Originally published in *The Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification (FATES/RV)*. Lecture Notes in Computer Science, Vol. 4262/2006, pp. 133-147, Springer-Verlag, August 2006, Seattle USA.

Paper II

Model-based Testing Using Scenarios and Event-B Refinements

Qaisar A. Malik, Johan Lilius and Linas Laibinis

Originally published in *M. Butler, C. Jones, A. Romanovsk and E. Troubitsyna (Eds.) Methods, Models and Tools for Fault Tolerance.* Vol. 5454/2009, *Lecture Notes in Computer Science*, pp. 177-195, Springer-Verlag, March 2009

Paper III

On Extending Scenario-based Test Case Generation Using Event-B Models

Qaisar A. Malik, Johan Lilius, Linas Laibinis and Dragoş Truşcan

Published (with additions) as *Requirement-driven Scenario-based Testing Using Formal Stepwise Development* in International Journal On Advances in Software. Vol. 3 Nr. 1 & 2, pp. 147-160, 2010.

Shorter version published as *Scenario-based Test Case Generation Using Event-B Models* in the Proceedings of 1st IEEE Intl. Conference on Advances in System Testing and Validation Lifecycle (VALID 2009), IEEE Computer Society, pp. 31-37, September 2009, Porto Portugal

On Extending Scenario-based Test Case Generation Using Event-B Models

Qaisar A. Malik, Johan Lilius, Linas Laibinis and Dragoş Truşcan
Turku Centre for Computer Science and Dept. of Information Technologies,
Åbo Akademi University, Turku, Finland.
Email: {Qaisar.Malik, Johan.Lilius, Linas.Laibinis, Dragos.Truscan}@abo.fi

Abstract

The paper presents an extension of our previously reported scenario-based testing approach based on formal models and user-provided testing scenarios. In our approach, the user provides a testing scenario on the level of an abstract model of the system under test. When the abstract model is refined to add or modify features, the corresponding testing scenarios are automatically refined to incorporate these changes. The resulting testing scenarios are unfolded into the test cases containing the required inputs and expected outputs. We use Event-B as our formal framework. The refinements of the system model and of its scenarios are later on transformed into Java code and respectively into JUnit test cases. In order to enable requirements traceability across the process requirements are linked to different parts of the specification, propagated at Java level, and associated to the JUnit tests.

1. Introduction

Testing is one of the most important and time-consuming activities in the software development process. Originally, testing has been performed manually, testers carefully considered the implementation under test and the designed test cases. As the software became more complex, the resulting test cases have grown in numbers and complexity. This naturally has lead to the need to automate the testing process. Today there exist several testing approaches that automate the testing process either completely or partially. These approaches try to achieve their goal by applying different means, i.e., code templates, scripts, formal and semi-formal software models etc.

Still, test automation still could not reduce the time taken to design the test cases, and thus a new approach came into place, namely *Model-Based Testing* (MBT) [1]. In MBT, models (typically behavioral) of the *system under test* (SUT) are used to automatically derive test cases based on selected coverage criteria. In these approaches, the generated tests do not distinguish between different parts of the system that might be more or less important for overall system correctness. In these cases, the whole process is based on the coverage criteria, such as transition coverage, state coverage or any other variation of these. Our scenario-based approach can be seen as an attempt to focus the testing process, by explicitly identifying important behavior of the system that should be tested.

We propose a methodology for scenario-based testing using formal models of the system and user-provided testing scenarios. These formal models and scenarios are mapped by the requirements. The formal models and scenarios are translated to Java and JUnit artifacts, respectively, while requirements are also propagated to the JUnit test cases. The work we present in this paper builds on our previous work [2], [3] on scenario-based testing, where we have used formal models of the SUT based on the Event-B formalism and we proposed a set of formal refinement techniques that can be used in the context of test generation.

Concretely our methodology proposes the following.

- inclusion of requirements in the formal specification process and propagation of requirements to tests;
- a method for identification of abstract test cases from formal scenario specifications;
- a method for generating Java templates of the system from sufficiently refined Event-B specifications;
- a method for generating JUnit tests from abstract test cases in CSP.

The organization of the paper is as follows. In Section 2, we give an overview of the model-based testing process and our scenario-based testing methodology. Section 3 presents extended guidelines for modeling of Event-B specifications and of testing scenarios along with requirements. In Section 4, we describe the technique for generation of implementation-templates for Java. In Section 5, we discuss JUnit test case generation. Section 6 contains some analysis and discussion. Finally, Section 7 concludes the paper.

2. Background

The Model-based Testing (MBT) process can be divided into following main phases [1], also shown in Figure 1

1) Modeling
2) Test Generation
3) Test Concretization
4) Test Execution
5) Test Result Analysis/Evaluation

In our previous work [2], we presented a scenario-based testing approach for generation of abstract test cases. The

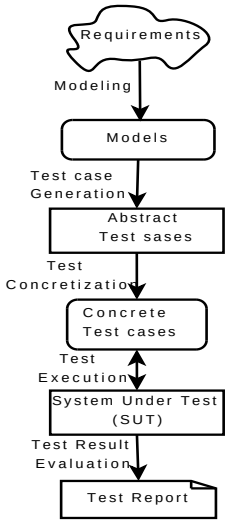


Figure 1. Model-based Testing Process

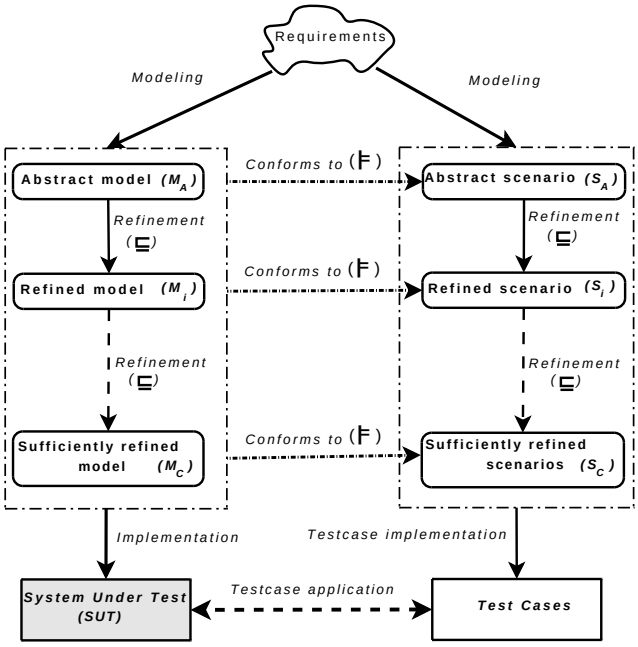


Figure 2. Overview of our scenario-based approach

aforementioned approach can be divided into two parallel development processes, one for formal development of models and other for development of testing scenarios. In the following, we give overview of the approach.

We use the Event-B formalism to model the behavior of the system as a state transition system. The development process starts with an abstract model and then gradually, by a number of refinement steps, a sufficiently detailed model is obtained. The implementation of the system under test (SUT) is an hand-coded implementation of this detailed model. Ideally, the final implementation should have been automatically generated from the detailed model. However, in practice, the final models still often lacks the low-level implementation details. Due to this *abstraction gap* between formal models and executable implementations, automatic generation of implementation code is not always possible. As a result, an implementation is often hand-coded, while consulting the formal models. Since the implementation is no longer *correct-by-construction*, the resulting implementation should be tested. Such tests could be designed by hand or generated automatically. We follow the latter approach.

For development of testing scenarios, we start from the requirements and gradually construct testing scenarios. The first abstract scenario is provided by the user. This scenario represents a valid behavior of the abstract model present on the same level of abstraction. In short, we say that the abstract model *conforms to* or formally *satisfies* the abstract scenario. Later on, we refine this abstract scenario along the refinement chain of the system models until a sufficiently detailed scenario is obtained. In fact, this detailed scenario represents an abstract test case. This process is presented graphically in Figure 2.

The formal models are refined manually based on the provided guidelines. The details of such refinement process are given in [2]. However, the generation of refined testing scenarios is automatic once an abstract scenario is provided by the user. This automatic process is merely a syntactic transformation from the abstract scenario S_A to the refined scenario S_i, while considering the controlled refinement steps involved in refining the abstract model M_A to the refined model M_i. Similarly, this process continues until we get a sufficiently detailed, concrete testing scenario S_C to which the model M_C conforms. The scenarios are represented as Communicating Sequential Process (CSP) [5] expressions. The refinement between scenarios is trace-refinement [6], while the conformance between models and scenarios is checked by the ProB [7] model checker. The satisfiability check is performed at each refinement level as was shown earlier in Figure 2.

More details about our scenario-based testing approach, modeling with Event-B, controlled refinements and case study example can be found in [2].

Overview of Event-B

The Event-B [4] is a recent extension of the classical B-method [8] formalism. Event-B is particularly well-suited for modeling event-based systems. The common examples of event-based systems are reactive systems, embedded systems, network protocols, web-applications and graphical user interfaces.

In Event-B, the specifications are written in Abstract Machine Notation (AMN). An abstract machine encapsulates state (variables) of the machine and describes operations

(events) on the state. A simple abstract machine has the following general form

MACHINE AM
VARIABLES v
INVARIANT I
EVENTS
$INITIALISATION \ = \ \ldots$
$E_1 \ = \ \ldots$
$\ldots$
$E_N \ = \ \ldots$
END

A machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLE** clause defines state variables, which are then initialized in the **INITIALISATION** event. The variables are strongly typed by constraining predicates of the machine invariant *I* given in the **INVARIANT** clause. The invariant defines essential system properties that should be preserved during system execution. The operations of event-based systems are atomic and are defined in the **EVENT** clause. An event is defined in one of two possible ways

$$E \ = \ \textbf{WHEN} \ g \ \textbf{THEN} \ S \ \textbf{END}$$

$$E \ = \ \textbf{ANY} \ i \ \textbf{WHERE} \ C(i) \ \textbf{THEN} \ S \ \textbf{END}$$

where *g* is a predicate over the state variables *v*, and the body *S* is an Event-B statement specifying how the variables *v* are affected by execution of the event. The second form, with the **ANY** construct, represents a parameterized event where *i* is the parameter and *C(i)* restricts *i*. The occurrence of the events represents the observable behavior of the system. The event guard (e.g., *g* or *C(i)*) defines the condition under which event is enabled.

Related Work

The jSynoPSys tool [9] performs scenario-based testing using symbolic animation of the B machines. This work defines a scenario-description language used to represent scenarios. However, authors do not provide any guidelines for the refinement of the specifications or scenarios. It is also not mentioned how scenarios will be transformed into executable test cases.

Nogueira et al. in [10] present a test generation approach based on the CSP formalism. The CSP models are constructed from use cases described in a pre-defined subset of natural language. The test scenarios are then incrementally generated as counter-examples for refinement verifications using a model checker. The main difference between their work and our approach is that we use Event-B to represent our system models and use CSP to represent testing scenarios. A model checker in our case is used to check the conformance between models and scenarios.

Stotts et al. in [11] describe a JUnit test generation scheme based on the algebraic semantics of Abstract Data Types (ADTs). The developer codes ADT in Java, while tests are generated for each ADT axiom. One of the advantages of this approach is that the formalism is hidden and the developer only needs to know Java to use this method. However, unlike our approach, in their case it would not be possible to mathematically prove any safety properties or to find deadlocks in the specifications.

In our earlier work [12], we presented the scenario-based testing approach for B models, where we designed an algorithm for constructing test sequences across different refinement [13] models. However, this algorithm is exponential in its nature thus limiting its practical applicability.

3. Event-B Models and Testing Scenarios

Our main goal in this paper is to generate executable test cases. In order to generate such concrete test cases, one needs to have enough information about the inputs and outputs of the system. In [2] we listed three basic types of refinement steps referred to as *controlled refinement*. In this section, we elaborate on these types of refinements by suggesting guidelines to incorporate information about the inputs and outputs of the system. These guidelines are used in a similar way for the development of both Event-B models and corresponding user scenarios.

In our scenario-based testing approach, we refine our models and scenarios in a stepwise manner (see Figure 2) based on user requirements. We will also elaborate on how the requirements can be attached to one or more of refinement steps as decided by the user.

3.1. Requirements

Usually the software systems are built according to informal requirements provided by user. The link between informal requirements and formal models is quite important in software development. We extend our previous scenario-based testing approach by introducing requirements in the specification process.

The requirements are used for creating the initial specification of the SUT and also for refining this specification on the next level of abstraction. The requirements are mapped to the Event-B models and to testing scenarios (see Figure 3).

In our approach, a stand-alone document specifying the requirements of the SUT in a structured manner is used. In this document, the requirements are specified using an ID and a text with the following structure:

`Requirement : REQ − ID`
`Text describing the requirement`

Hierarchy of the requirements is implemented using the requirement ID. For instance, requirement `REQ − 1.1` is a sub-requirement of requirement `REQ − 1`.

Throughout this paper we will use small examples from a *Hotel Booking System*. The system should allow the user to search for a room in the room database (REQ − 1), to reserve the room (REQ − 2), to allow him to pay for the reserved room (REQ−3) or to cancel an existing reservation (REQ−4). The requirement REQ − 1 is described as

Requirement : REQ − 1
The system should be able to find a room of given type if it is available in the database and connection to the database is successfully established. In case of failed connection, an exception is reported.

Each requirement can be divided into several sub-requirements. For instance, (REQ − 1.1) and (REQ − 1.2) are given in the following.

Requirement : REQ − 1.1
The system should be able to find a room of given type if it is available in the database and connection to the database is successfully established.

Requirement : REQ − 1.2
The system should return an error message if connection to the database is not established successfully.

The requirement (REQ − 1.1), is further divided as

Requirement : REQ − 1.1.1
The system should be able to accept room type as an input.

Requirement : REQ − 1.1.2
The system should be able to connect to the database.

Requirement : REQ − 1.1.3
The system should be able to retrieve results.

These sub-requirements serve as basis for refinements in the Event-B model. A plug-in for the RODIN platform [14], the Requirement plug-in [15], is used for mapping requirements between the requirement document and the model. In the Requirement plug-in, a parser parses the requirement document and lists individual requirements. Then, any requirement can be selected to be mapped to one or more Event-B elements. This mapping information is stored in a mapping file. Similarly, a separate mapping file is created for storing the mapping between requirements and scenarios.

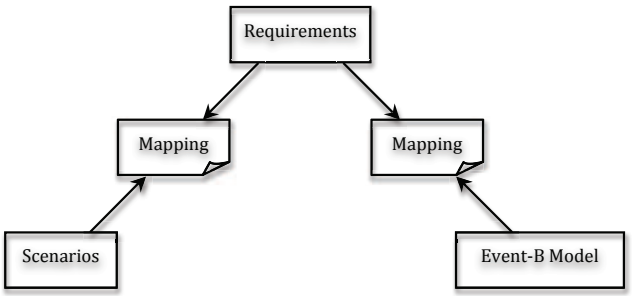


Figure 3. Mapping Requirements to Specifications and Scenarios

3.2. Using EventB for system specification

In our approach, we create formal descriptions of the SUT starting from the requirements as shown in Figure 2. Subsequent refinements of the specification are preformed based on the sub-requirements of a given requirement. In this section, we discuss our controlled method for modeling and refining the system specification. In order to facilitate the refinements of our EventB specifications, we introduce a classification of the event types and a new *logical unit* concept.

3.2.1. Classification of Events. In order to identify the inputs and the outputs of the system, we classify the events of our Event-B models as of *input*, *output*, and *internal* types.

Definition 1: The Events. Set of all events in the system, denoted by Σ, is divided into following subsets of:

- *Input* events denoted by ε^I
- *Output* events denoted by ε^O
- *Internal* events denoted by ε^τ

□

The *input events*, ε^I, accept inputs from user or environment. Apart from their input behavior, these events may take part in the normal functioning of the system. However, the input events do not produce externally visible output. The *output events* ε^O produce the externally visible outputs. Finally, the *internal events* do not take part in any input/output activity. These events however, may produce intermediate results used by the events in ε^I and ε^O. The motivation of this classification is explained in next section, where we divide our system into logical functional units.

3.2.2. Logical Units. As we develop our system in a stepwise manner, the main functional units of a system are already identified on the abstract level. Each of these abstract functional units are modeled as a separate logical unit, called *IOUnit*, in our Event-B models.

Definition 2. An IOUnit, U, consists of a finite sequence of events and has the following form.

$$U =< \varepsilon^I, \varepsilon^{\tau+}, \varepsilon^O >$$

Here ε^I and ε^O denote the input and output events respectively, and $\varepsilon^{\tau+}$ represents one or more occurrences of *internal* events.

□

It can be observed from the above definition that an *IOUnit* consists of the sequence of events occurring in such an order that the first event in the unit is always an *input* event and the last event is always an *output* event, with possibly one or more *internal* events in between. Moreover, an *IOUnit* can not contain more than one input or output event.

An *IOUnit* takes input and produces output, as the presence of the input and output events indicates. The classification of events, defined in the previous section, helps us in identifying the inputs and outputs of each unit, and when combined, of the whole system. The motivation for this approach is the following. The developer of the SUT may decide to implement the system independently of the structure of an Event-B model. Indeed, it is sometimes hard to construct the strict one-to-one mapping between the events of the model and corresponding programming language units. For example, two events in a model can be merged to form one programming-language operation, or the functionality of an event in the model may get divided across multiple operations or classes in the implementation. However, for successful execution of the system, the interfaces of the model and implementation, i.e., the sequence of the inputs and outputs, should remain the same.

3.2.3. Example. Reserving a room in such the hotel booking system can be modeled as a sequence of events that occur in a specific order. On the abstract level, we may have only a few events, representing some particular functionalities of the system. For example, if we model requirements $\texttt{REQ} - 1$ to $\texttt{REQ} - 4$, each top-level user requirement will be implemented as one IOUnit. Consequently, there are four main IOUnits namely, *Finding* a room, *Reserving* it, *Paying* for it, and *Canceling* a reserved room. After we structure our model according to the guidelines described in Section 3.2.1, the resulting events and their sequence of execution can be seen in Figure 4(a).

As it can be observed, the main functional events are wrapped with the input and output events. For example, the *Find* event is wrapped around with the *InputForFind* and *OutputForFind* events, where *InputForFind* and *OutputForFind* are the input and output events, respectively.

Within an IOUnit, we treat our main functional events as *internal events* (e.g., *Find, Reserve, Pay* and *Cancel*). Such events can be further refined, in one or more steps,

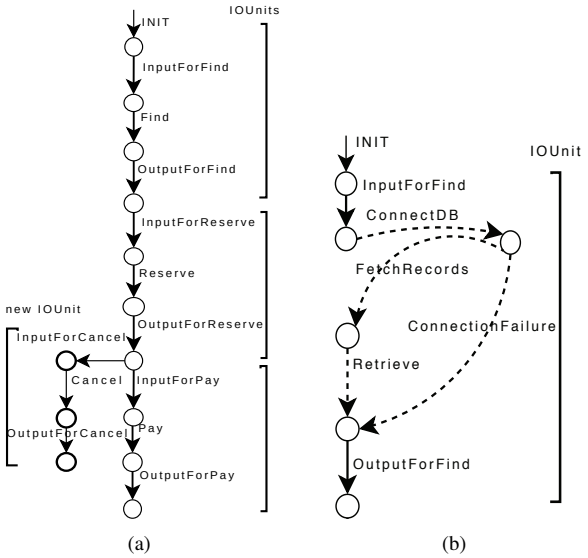


Figure 4. (a) Abstract System (b) Refined System

consequently adding more *internal* events within the input-output unit. The refinement is performed according to the sub-requirements of the requirement that was the source of the IOUnit. For instance, the *Find* IOUnit in Figure 4(a) has been refined by applying successive refinements based on the requirement $\texttt{REQ} - 1$ and its sub-requirements, introduced earlier in Section 3.1, into four *internal* events depicted graphically with dashed line pattern in Figure 4(b).

The complete Event-B specifications of this example have been developed and proved using the RODIN [14] platform. In the final refined system, there was a total of 42 proof obligations. Out of these, 38 proof obligations were automatically discharged by the tool, while the remaining 4 needed manual assistance.

3.3. Testing Scenarios

We mentioned before that we use Communicating Sequential Processes (CSP) [5] to represent testing scenarios. We define a testing scenario as a finite sequence of events occurring in some particular order. Since we have grouped the events in the form of logical IOUnits, our scenarios will also include a finite sequence of IOUnits. This means that the scenarios will include the same events as in the corresponding Event-B model. However, the scenarios must follow the same rules that were set for constructing IOUnits in the previous section, i.e.,

1) The first event in the scenario is always an *input* event;
2) The last event in the scenario is always an *output* event;
3) There can not be two input-type events in the sequence without any output event in between them, i.e., the following sequence in a CSP expression is not allowed;

$$< \cdots \rightarrow \varepsilon^I_k \rightarrow \varepsilon^I_{k+1} \rightarrow \cdots >$$

4) There can not be two output-type events in sequence without any input event in between them, i.e., the following sequence is also not allowed.

$$< \cdots \rightarrow \varepsilon^O_k \rightarrow \varepsilon^O_{k+1} \rightarrow \cdots >$$

Since the scenarios are defined on the abstract level, they lack details about the system inputs and outputs. The input details can be identified from the input event(s) of each IOUnit. For example, if an input event reads three input variables then these three variables become the inputs for the unit that the input event belongs to. The details about the inputs can be retrieved from the Event-B model since the model specifies the type, initial value and invariant properties for all variables.

The expected outputs are generated after the model is animated using the ProB model checker [7]. For a given input of a test case, the ProB can animate the model and return the result, which is then saved as the *expected* output of the test case. This expected output can be then used to compare the values while testing the real implementation. The ProB model checker can only produce output values based on the available abstract values. For example, to test whether a room is available in the *Hotel Booking System*, ProB can check the expected result for a pre-defined set of inputs, while in the actual implementation this result might be retrieved from the database. Therefore, we need to define a mapping relation between the abstract and concrete data types. At the moment this mapping is provided manually. However, it is possible to automate its generation for the commonly used types, e.g., boolean and integers.

3.3.1. Example. In the case of the previously discussed *Hotel Booking System* example, there can be many possible testing scenarios. For example, if we want to test the *room finding*, *reservation* and *paying* functionality, the corresponding abstract scenario expressed as a CSP expression would be as follows.

$$\begin{aligned}
&\mathtt{S}_{0(\mathtt{A})} = \mathtt{InputForFind?roomType} \rightarrow \mathtt{Find} \rightarrow \\
&\quad \mathtt{OutputForFind!}(\mathtt{roomId}, \mathtt{anyException}) \rightarrow \\
&\quad \mathtt{InputForReserve?roomId} \rightarrow \mathtt{Reserve} \rightarrow \\
&\quad \mathtt{OutputForReserve!reserveId} \rightarrow \\
&\quad \mathtt{InputForPay?reserveId} \rightarrow \mathtt{Pay} \rightarrow \\
&\quad \mathtt{OutputForPay!payId}
\end{aligned}$$

After a number of successive refinements of event *Find*, we achieve the following scenario. For keeping the example simple, we only show the refinement of *Find* event which is also shown graphically in Figure 4(b).

$$\begin{aligned}
&\mathtt{S}_0 = \mathtt{InputForFind?roomType} \rightarrow \mathtt{ConnectDB} \rightarrow \\
&((\mathtt{FetchRecords} \rightarrow \mathtt{Retrieve}) \sqcap \mathtt{ConnectionFailure})) \\
&\rightarrow \mathtt{OutputForFind!}(\mathtt{roomId}, \mathtt{anyException}) \rightarrow \\
&\mathtt{InputForReserve?roomId} \rightarrow \mathtt{Reserve} \rightarrow \\
&\mathtt{OutputForReserve!reserveId} \rightarrow \\
&\mathtt{InputForPay?reserveId} \rightarrow \mathtt{Pay} \rightarrow \\
&\mathtt{OutputForPay!payId}
\end{aligned}$$

where $\sqcap$ is the internal choice operator in CSP. The variable `roomType` is the input for this IOUnit, whereas `roomId`, `anyException` are possible outputs. The variable `anyException` specifies if there was any exception, e.g., a connection failure.

Often, the subsequent event depends on the results of the previous ones. For example, the event `Reserve` takes `roomId` as an input from the previous event. It can be noticed that the refinement of the `Find` event has created two branches, one leading to successful case and the other to a database connection failure exception. When the above scenario is checked for conformance with the ProB model checker, it will be found that one can not proceed to `Reserve` if an exception occurred at the previous step. Therefore, this scenario will be split into two scenarios $\mathtt{S}_0$ and $\mathtt{S}_1$ given in the following.

$$\begin{aligned}
&\mathtt{S}_0 = \mathtt{InputForFind?roomType} \rightarrow \mathtt{ConnectDB} \rightarrow \\
&\mathtt{FetchRecords} \rightarrow \mathtt{Retrieve} \rightarrow \\
&\mathtt{OutputForFind!}(\mathtt{roomId}, \mathtt{anyException}) \rightarrow \\
&\mathtt{InputForReserve?roomId} \rightarrow \mathtt{Reserve} \rightarrow \\
&\mathtt{OutputForReserve!reserveId} \rightarrow \\
&\mathtt{InputForPay?reserveId} \rightarrow \mathtt{Pay} \rightarrow \\
&\mathtt{OutputForPay!payId}
\end{aligned}$$

$$\begin{aligned}
&\mathtt{S}_1 = \mathtt{InputForFind?roomType} \rightarrow \mathtt{ConnectDB} \rightarrow \\
&\mathtt{ConnectionFailure} \rightarrow \\
&\mathtt{OutputForFind!}(\mathtt{roomId}, \mathtt{anyException})
\end{aligned}$$

These scenarios, when sufficiently refined, are transformed into JUnit tests which will be discussed later in Section 5.

In the next section, we will discuss how Event-B model is used to generate an implementation template in Java.

4. Generating Java Implementation Templates

Once developed, we use the Event-B models of the SUT to generate Java implementation templates. We start by translating a (sufficiently refined) Event-B model into a Java class. As a result, Event-B events are translated to the corresponding Java methods. For our *Hotel Booking System* example, the excerpts of the respective Event-B machine and its implementation template are shown as follows.

An operation in an Event-B specification consists of two parts. The first part contains the pre-condition(s) for the event operation to be enabled, while the second part consists of the actions that the operation performs. For every event in an Event-B model, we create two separate methods in the corresponding Java implementation representing the pre-conditions and actions respectively. The first method, which contains the pre-conditions of an event, returns the evaluation result in the form of a *boolean* value. The name of this method is pre-fixed with the string "guard_". The

MACHINE BookingSystemRef1
REFINES BookingSystem
SEES BookingContext
VARIABLES
roomType
...
INVARIANTS
...
EVENTS
Initialisation
act5 : roomType := Null_roomType
...
Event InputForFind ≙
Refines InputForFind
any
tt
where
grd1 : tt ∈ RTYPES
then
act1 : roomType := tt
act2 : inputForFindCompleted := TRUE
end
...
END

second method encapsulates the actions of the event. For example, for the *InputForFind* event from our *Hotel Booking System* example, the Java implementation methods are given in the Listing 1. As one can notice, the requirements attached to different IOUnits in Event-B are preserved during the transformation and included in the generated template (see line 19 of Listing 1).

```
1  public class HotelBookingSystem {
2
3      //class-level variables
4      public String roomType;
5      ....
6
7      public HotelBookingSystem(){
8          //initialization     ...
9      }
10
11     /* PreConditions/Guards for InputForFind event
           */
12     private boolean guard_inputForFind(String
           roomType){
13         return (roomType != null);
14     }
15
16     /* Implementation method for InputForFind
           event*/
17     public boolean inputForFind(String roomType)
18             throws PreConditionViolatedException{
19        //REQ-1.1.1
20
21         boolean inputForFindCompleted = false;
22         if(guard_inputForFind()){
23
24             //actions ...
25
26             this.roomType = roomType;
27             inputForFindCompleted = true;
28         }
29         else{
30             throw new
31             PreConditionViolatedException("For
                   inputForFind");
32         }
33         return inputForFindCompleted;
34     }
35
36     //more Implementation methods for events
37     ....
38 }
39
40 class PreConditionViolatedException extends
       Exception{
41
42     public PreConditionViolatedException(String
           mesg){
43         super(mesg);
44     }
45 }
```

Listing 1. Implementation template example

Each Java implementation method, representing an Event-B event, first evaluates its pre-condition(s) by calling its "guard_" method. If the pre-conditions are evaluated to *false* then the exception `PreConditionViolatedException` is raised, otherwise the actions of the corresponding event are executed. The variables of an Event-B machine are translated into the corresponding class variables in Java. The type information for these variables can be retrieved from the *invariant* clause of the Event-B machine. We assume that a mapping relation between data types in Event-B and Java is provided by the user. While most of the Java code can be automatically translated from Event-B constructs, the user can add more code statements according to his/her requirements. This means that the generated class actually constitutes a Java template.

In the next section, we will discuss how testing scenarios are translated into JUnit test cases.

5. Generating JUnit test cases from Scenarios

In Section 4, we presented the guidelines for generating implementation templates for Java. Once such a template is generated, we can generate the corresponding executable test cases from the scenarios. These test cases can be represented as Java Unit Testing (JUnit) [16] and TestNG [17] test methods.

Since our Event-B events are now presented as sequences of *IOUnits*, we write JUnit test cases to test these *IOUnits*. The $Find$ IOUnit from scenario $\mathtt{S}_0$ is represented as an abstract test case $\mathtt{T}_0$ as given in the following.

$$\mathtt{T}_0 = \mathtt{InputForFind?roomType} \rightarrow \mathtt{ConnectDB} \rightarrow \mathtt{FetchRecords} \rightarrow \mathtt{Retrieve} \rightarrow \mathtt{OutputForFind!(roomId, anyException)}$$

For scenario $\mathtt{S}_1$, the abstract test case $\mathtt{T}_1$ would be expressed as following.

$T_1 = \texttt{InputForFind?roomType} \rightarrow \texttt{ConnectDB} \rightarrow$
$\texttt{ConnectionFailure} \rightarrow$
$\texttt{OutputForFind!(roomId, anyException)}$

For each of the test cases T_0 and T_1, a separate JUnit test method is implemented. The JUnit test method for T_0 is shown in the Listing 2. In a similar way, JUnit test cases are generated for each IOUnit in the scenario.

```
1  public class HotelBookingSystemTest {
2
3      HotelBookingSystem bSys;
4      ...
5
6      @Before
7      public void setUp() throws Exception {
8          bSys = new HotelBookingSystem();
9      }
10
11     @Test(dataprovider = ''roomTypes'')
12     public final void T0(String roomType){
13         //REQ-1.1
14         try{
15             boolean v1,v2,v3,v4,v5;
16             v1 = v2 = v3 = v4 = v5 = false;
17
18             //calling methods of IOUnit
19             v1 = bSys.inputForFind(roomType);
20             v2 = bSys.connectDB();
21             v3 = bSys.fetchRecords();
22             v4 = bSys.retrieve();
23             v5 = bSys.outputForFind();
24
25             //assert statements (verdict)
26             assertTrue(''Successful completion'',
27                 v1 && v2 && v3 && v4 && v5);
28
29             assertTrue(bSys.resultSet.size() > 0);
30             assertTrue(bSys.anyException == false)
                   ;
31         }
32         catch(PreConditionViolatedException e){
33             fail(e.getMessage());
34         }
35     }
36     // TestNG method to provide input data
37     @DataProvider(name = ''roomTypes'')
38     public Object[][] createAllRoomTypes(){
39
40         return new Object[][]{
41             new Object[] {''Single''},
42             new Object[] {''Double''};
43         }
44      }
45 }
```

Listing 2. JUnit/TestNG Test method for T_0

In the test case example shown in Listing 2, there is only one input parameter, i.e., `roomType`. However, in practice, there can be more than one input parameters. Generating all possible values for each parameter and then making all possible combinations of these parameters values may result in combinatorial explosion. In order to handle this problem, the input space partitioning [18] approach is used for test case generation. Information about each input variable is retrieved from the *invariant* clause and the *pre-condition* part of the *input* event. The *pre-conditions* and *invariant* clauses specify the type and possible restrictions (value ranges) for each variable. Using this information, the input space for each parameter is divided into equivalent partitions. Then from each partition, one value is selected to represent the whole partition. Combining the values of different variables from different partitions reduces the total number of input combinations needed for testing. These combinations are provided to JUnit test cases using the *@DataProvider* method of the TestNG framework [17]. An example of such a method can be observed in Listing 2.

If a scenario involves multiple *IOUnits* in a sequence and JUnit test case for that sequence is desired, then JUnit test also includes calls to the relevant implementation methods of the the *IOUnit* involved. Moreover, the JUnit assert statements are also appended in the test case.

It is important that the system requirements are propagated from scenarios to JUnit test cases. Although, requirements can be attached at any one or more of the refinement steps, however, it is important that the requirements are mapped to the sufficiently refined models and scenarios. This is because the sufficiently refined models and scenarios (at this stage called abstract test cases) are transformed into implementation constructs in Java. The resulting JUnit test case would also carry the information about requirements and upon their execution, it would be possible to trace which requirements have or have not been fulfilled.

The requirement $\texttt{REQ} - 1.1$ and $\texttt{REQ} - 1.2$ are attached (see Figure 5) to the above mentioned test cases, T_0 and T_1 respectively, and this information is appended as Java comments in resulting JUnit test cases as can be seen on line 13 in the Listing 2.

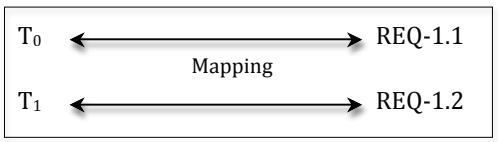


Figure 5. Mapping for informal requirements to abstract test cases

In the next section, we analyze our testing approach and discuss some related issues.

6. Discussion

The presented test generation process produces test cases in JUnit, which is a well-known and widely used testing framework. The test cases are generated according to the user provided scenarios. More scenarios the user provides, the more code coverage we are likely to achieve. There are several good coverage measuring tools available that can be used with the generated test suites. We have tried EclEmma [19], which is a freely available open source Java

code coverage tool available for Eclipse, and found it quite easy to use with our approach. With EclEmma, it is also possible to generate the test execution and coverage analysis reports.

Furthermore, our approach has the distinguishing advantage that it also accommodates those changes which can not categorized and proved as formal refinement. Referring back to Figure 2, in some cases the model M_i may contain some extra functionalities or features, such as the incorporated fault-tolerance mechanisms, which were omitted or out of scope of the scenario S_A. These *extra features*, denoted by S_{EF}, can be added in the scenario S_i manually. The modified scenario $S_i \cup S_{EF}$ must be checked, by means of the ProB model checker, to satisfy the model M_i. We can then follow the same refinement process, now starting with $S_i \cup S_{EF}$, until we get a sufficiently refined scenario at level of the final model M_C.

Our approach also describes how one can generate Java implementation templates and the corresponding JUnit test cases. However, if for some reason, the user does not want to use the generated template, s/he can still use the JUnit test generation part to test his/her own implementation, provided that s/he has implemented the system keeping the operation interfaces consistent with the already generated JUnit test cases.

At the moment, we do not support translation of more complex pre-condition and invariant expressions from Event-B to Java. Namely, the existential and universal quantifiers are not covered. However, this can be achieved by using an approach similar to the one used in JML [20].

We do not explicitly support testing for *negative* scenarios i.e., the behavior that should not exist in the SUT. However, this kind of testing can also be accommodated if we model such *negative* behavior in our Event-B models as events and then provide testing scenarios covering those events. In order to show correctness, the JUnit tests, generated from these negative scenarios, should fail when applied on SUT.

7. Conclusions

In this paper, we presented a model-based testing approach using user-provided testing scenarios. These scenarios are first validated using a model checker and then used to generate test cases. Additionally, we have provided the guidelines for stepwise development of formal models and automatic refinement of testing scenarios. We also proposed an approach to generate Java language implementation templates from Event-B models. The abstract testing scenarios can then be used to generate executable JUnit test cases. Optionally, user can map informal requirements to the formal model and testing scenarios at different refinement steps. This mapping of informal requirements is extended till concrete test cases so that upon test case failure, these unfulfilled requirements can be back-traced into the model.

We believe that our approach is very scalable. It can help developers and testers to automatically generate large number of executable test cases. Generating these test case by hand would be very laborious and error-prone process.

As future work, we aim at providing graphical representation for the testing scenarios and their refinements. Moreover, at the moment, the mapping between abstract and concrete data types needs to be provided manually by the user. An automatic translation would be very helpful and time-saving in this respect.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing*. Morgan Kaufmann Publishers, 2006.

[2] Q. A. Malik, J. Lilius, and L. Laibinis, "Model-Based Testing Using Scenarios and Event-B Refinements," in *Methods, Models and Tools for Fault Tolerance, LNCS Vol. 5454*. Springer-Verlag, 2009, pp. 177–195.

[3] Q. A. Malik, J. Lilius, and L. Laibinis, "Scenario-Based Test Case Generation Using Event-B Models," in *International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*. IEEE Computer Society, 2009, pp. 31–37.

[4] J.-R. Abrial, "A System Development Process with Event-B and the Rodin Platform," in *ICFEM*, 2007, pp. 1–3.

[5] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[6] A. Roscoe, *The theory and practice of concurrency*. Prentice Hall, 1998 amended 2005.

[7] M. Leuschel and M. Butler, "ProB: A model checker for B." Proc. of FME 2003, Springer-Verlag LNCS 2805, pages 855-874., 2003.

[8] J.-R. Abrial, *The B-Book*. Cambridge University Press, 1996.

[9] F. Dadeau and R. Tissot, "jSynoPSys – A Scenario-Based Testing Tool based on the Symbolic Animation of B Machines," *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 2, pp. 117–132, 2009.

[10] S. Nogueira, A. Sampaio, and A. Mota, "Guided Test Generation from CSP Models," in *ICTAC*, 2008, pp. 258–273.

[11] P. D. Stotts, M. Lindsey, and A. Antley, "An Informal Formal Method for Systematic JUnit Test Case Generation," in *XP/Agile Universe*, 2002, pp. 131–143.

[12] M. Satpathy, Q. A. Malik, and J. Lilius, "Synthesis of Scenario Based Test Cases from B Models." in *FATES/RV*, 2006, pp. 133–147.

[13] R.-J. Back and J. von Wright, "Refinement Calculus, Part I: Sequential Nondeterministic Programs," in *REX Workshop*, 1989, pp. 42–66.

[14] “Rigorous Open Development Environment for Complex Systems,” iST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.

[15] “Requirement Management Plug-in for Rodin Platform,” home page : http://wiki.event-b.org/index.php/Category:Requirement_Plugin.

[16] “JUnit 4,” http://www.junit.org.

[17] C. Beust and H. Suleiman, *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley, 2007, http://www.testng.org/.

[18] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[19] “EclEmma - Java Code Coverage for Eclipse,” http://www.eclemma.org/.

[20] G. T. Leavens and A. L. Baker, “Enhancing the Pre- and Postcondition Technique for More Expressive Specifications,” in *In FM99: World Congress on Formal Methods*. Springer, 1999, pp. 1087–1106.

Paper IV

Using UML Models and Formal Verification in Model-Based Testing

Qaisar A. Malik, Dragoş Truşcan and Johan Lilius.

Originally published in *In Proceedings of 17th IEEE Intl. Conference on Engineering of Computer-Based Systems (ECBS 2010)*, IEEE Computer Society, pp. 50-56, March 2010, Oxford UK.

2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Using UML Models and Formal Verification in Model-Based Testing

Qaisar A. Malik, Dragoş Truşcan and Johan Lilius
Turku Centre for Computer Science (TUCS)
Department of Information Technologies, Åbo Akademi University, Turku, Finland.
Email: {Qaisar.Malik, Dragos.Truscan, Johan.Lilius}@abo.fi

Abstract

In this paper we present a model-based testing approach where we integrate UML, UML-B and the Qtronic test generator tool, with the purpose of increasing the quality of models used for test generation via formal verification. The architectural and behavioral models of the system under test (SUT) are specified in UML and UML-B, respectively. UML-B provides UML-like visualization with precise mathematical semantics. UML-B models are developed in a stepwise manner which are then automatically translated into Event-B specifications that can be proved using theorem provers. Once the formal models are proved, they are transformed into QML which is a modeling language used by the test generation tool.

Index Terms

UML-B; UML based testing; Model-based testing; Event-B

1. Introduction

During recent years, Model-based Testing (MBT) has gained noticeable popularity due to its success in automated testing. The term *Model-based testing* refers to the kind of testing where tests are generated from models [1, page 6]. The main purpose of software testing is to find defects and increase the confidence in a given system implementation, by checking weather the implementation conforms to the specifications.

There are several established approaches and available tools for generating tests from behavioral models of the system under test (SUT) (see [1] for details). However, complex systems are often described using several perspectives like architectural, behavioral, data etc., which may also aid the testing process. One such approach of using different perspectives of the system for MBT has been discussed in [2]. The mentioned approach uses several system models, modeled using the Unified Modeling Language (UML) [3], and transforms these into input for an automated test design tool named Qtronic [4].

Despite of its extensive use in both industry and academia, one commonly stated argument against using UML is about the lack of precise formal semantics. In order to improve quality of the models, consistency checking and validation is performed by developing various rules and guidelines [5], mainly by using OCL [6]. However, there still exists a need for formal verification for the behavioral part of the system, which by improving the quality of the generated models will eliminate unnecessary failed test cases due to inconsistencies in the models used for test generation.

This paper focuses on the aspect of how to incorporate formal verification in this existing model-based testing process. For formal verification, we use UML-B which is a new formal modeling notation combining UML with the B-method [7], thus providing UML-like visualization with precise mathematical semantics.

The organization of this paper is as follows. Firstly, some background theory and concepts about UML-B, Qtronic and our model-based testing approach is presented in Section 2. In Section 3 and 4, we describe, with a case study, how formal verification is incorporated in our existing model-based testing process. Section 5 contains references to some related work in this area. Finally, Section 6 provides an evaluation of our approach and concludes the paper.

2. Background

2.1. Model-Based Testing Process

The earlier work, that we are extending here, is on using UML models to specify the SUT, by using data models, test configuration models, domain models and state-machines.

To show details of this methodology, we use excerpts from a telecommunication case study modeling a *Mobile Switching Server* (MSS), which will be used as the SUT. The MSS is the main element of 2G (2nd Generation) and 3G (3rd Generation) networks. The role of MSS is to connect calls between mobile phones and fixed networks. For this case study, we model *Location Update* feature of MSS which handles location information of mobile phones at the time of call setup and during the calls. The communication between MSS and MS (Mobile Subscriber) is performed by exchanging messages that is why this work is tailored for message-based communicating systems.

The class diagram (Fig. 1), as domain model, is used to represent components of the domain and how these com-

978-0-7695-4005-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ECBS.2010.13

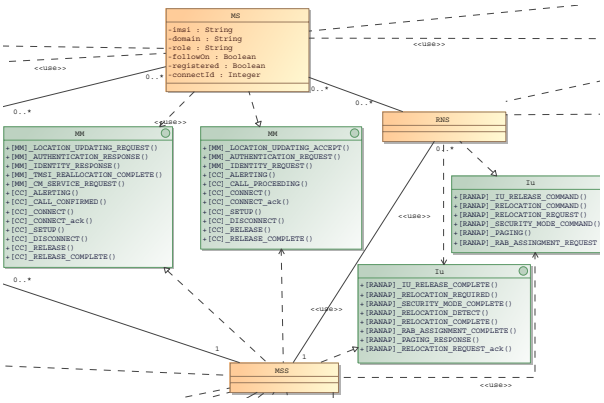


Figure 1. Class diagram representing a Domain Model

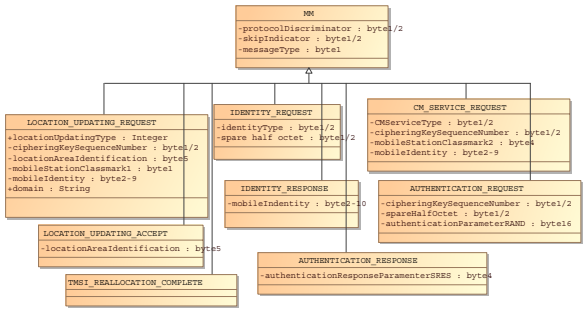


Figure 2. Message declaration in UML

ponents are connected via interfaces. Domain components communicate with each other using messages belonging to various protocols. The messages sent and received on each protocol level are modeled separately using `interface` classes (e.g., `MM`).

Since test cases will be generated for the MSS component, which will farther on regarded as the system-under-test (SUT), a description of the data, the messages, sent and received by the system is needed. This data is modeled explicitly via class diagrams referred as *data models*. A data model depicts each message type as a class where relevant fields or parameters of the messages are represented as class attributes. The message definitions are structured based on corresponding protocols. Figure 2 presents an example of a UML data model from our case study.

The behavioral part is specified by UML state machine diagrams. In previous work it has been discussed how the UML models are created [2], validated [5] and then transformed to QML [8] to be used for automated test generation in the Qtronic tool. Here in this paper, we emphasize more on incorporating formal verification in this modeling and test generation process.

2.2. Modeling with QML - The Qtronic tool

Conformiq Qtronic [4] is a tool for model-driven test case design. It generates test cases from the specification of the SUT. The *Qtronic Modeling Language* (QML) is a Java-like modeling language used by Qtronic. The behavior of the SUT can be specified in Qtronic using state machines, whereas the QML used as action language and for specifying the test configuration. The SUT is specified as a `class` that can have attributes and methods. In Qtronic, messages sent or received by the SUT are defined as records, that is user-defined types that can contain variables, methods, operators and nested types. The messages sent and received by SUT are described by the `Inbound` and `Outbound` ports. In QML, a state-machine, describing the SUT behavior, runs as a separate thread. However, multiple threads of the SUT can be executed facilitating testing of concurrent behavior.

2.3. Modeling with UML-B

2.3.1. UML-B. UML-B [9], is a graphical formal modeling notation that combines UML and Event-B method. UML-B is similar to UML but has its own meta-model. UML-B provides tool support that includes a drawing tool and a translator to generate Event-B models. The tool support is provided in the form of a plug-in for the RODIN [10] platform which is an Eclipse-based formal development framework for Event-B. By using UML-B one can graphically model various aspects of a system using class diagrams and state machines. One can also attach, graphically, formal constructs like invariants and theorems to the two diagram types. Since UML-B uses the precise mathematical semantics of Event-B, everything in graphical model is automatically translated into Event-B specifications. In the following, we give an overview of Event-B method.

2.3.2. Overview of Event-B. The Event-B [11] is a recent extension of the classical B-method [7] formalism. Event-B is particularly well-suited for modeling event-based systems. The common examples of event-based systems are reactive systems, embedded systems, network protocols, web-applications and graphical user interfaces.

In Event-B, the specifications are written in Abstract Machine Notation (AMN). An abstract machine encapsulates state (variables) of the machine and describes operations (events) on the state. A simple abstract machine has following general form

MACHINE AM
VARIABLES v
INVARIANT I
EVENTS
$INITIALISATION \ = \ \ldots$
$E_1 \ = \ \ldots$
$\ldots$
$E_N \ = \ \ldots$
END

A machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLE** clause defines state variables, which are then initialized in the **INITIALISATION** event. The variables are strongly typed by constraining predicates of the machine invariant *I* given in the **INVARIANT** clause. The invariant defines essential system properties that should be preserved during system execution. The operations of event-based systems are atomic and are defined in the **EVENT** clause. An event is defined in one of two possible ways

$$E \quad = \quad \textbf{WHEN} \ \ g \ \ \textbf{THEN} \ \ S \ \ \textbf{END}$$
$$E \quad = \quad \textbf{ANY} \ \ i \ \ \textbf{WHERE} \ \ C(i) \ \ \textbf{THEN} \ \ S \ \ \textbf{END}$$

where *g* is a predicate over the state variables *v*, and the body *S* is an Event-B statement specifying how the variables *v* are affected by execution of the event. The second form, with the **ANY** construct, represents a parameterized event where *i* is the parameter and *C(i)* restricts *i*. The occurrence of the events represents the observable behavior of the system. The event guard (e.g., *g* or *C(i)*) defines the condition under which event is enabled.

3. Combining UML and Event-B

The main goal of this work is to increase the quality of the models used for test generation. UML is used for modeling architectural and static aspects of the SUT, whereas UML-B is used to formally verify behavioral aspects of the system. These models are transformed into QML. Qtronic generates test cases from these models which are then executed on SUT. The *online* testing mode of Qtronic has been used in our approach, where Qtronic generates tests and applies them on-the-fly against the SUT. Figure 3 depicts the overall process.

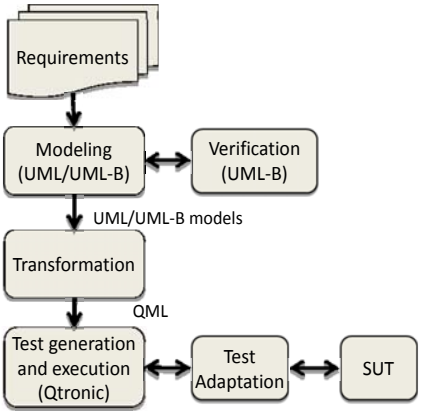


Figure 3. The overall process

3.1. Modeling Behavior in UML-B

In this work, UML-B is mainly used to model state-machines. In order to handle complexity, we use a step-wise development approach. First, an abstract model is specified which is further refined in the next steps to add more details. The graphical models in UML-B are automatically translated into Event-B textual representation. The generated Event-B specifications are represented by modules named Machines. The added-value of using UML-B is that it is possible to add formal first-order logic expressions as the guards of the transitions of state-machines and invariants of the system. The Event-B specifications are proved by theorem provers at each refinement step. At the moment, UML-B only supports refinement of classes and state-machines. Our new approach is applied and exemplified with excerpts from the same telecommunications case study discussed in Section 2.1.

As described earlier, we develop state machines in a stepwise manner. In the first step, an abstract state machine is defined to model the Location Update functionality of our SUT. The state machine (Figure 4(a)) has four main states, namely *Idle*, *Authentication*, *Ciphering* and *ReleaseChannel*, representing the main functionality of the system abstractly. In a later refinement step, we add detailed information about each functionality by introducing sub-state machines as shown in Figure 4(b). The sub-state machine for *ReleaseChannel* state is shown in Figure 5.

3.2. Transformation to QML

Once a sufficiently refined and formally proved model is obtained, we proceed to the next phase, transforming it into QML. The overall transformational process, for individual constructs, for UML, UML-B and QML is shown in the Figure 6.

As described earlier, the behavior of the SUT is described in the form of state machines. There are two main purposes for modeling behavior using state machines. First, by using UML-B state machines we formally verify behavioral properties of SUT specification. Second, Qtronic tool expects the behavior of SUT in the form of state machines, therefore, we transform the verified state and sub-state machines from UML-B to Qtronic.

In QML, a transition on the state machine may consists of the following:

$$[trigger][guard]/[actions]$$

The triggers are implemented by messages received on a certain port. The guard is a boolean condition, often a comparison of values or fields. The actions are the methods of the SUT class definition which represent sending of a message. All of these three constructs that constitute a transition are optional in QML.

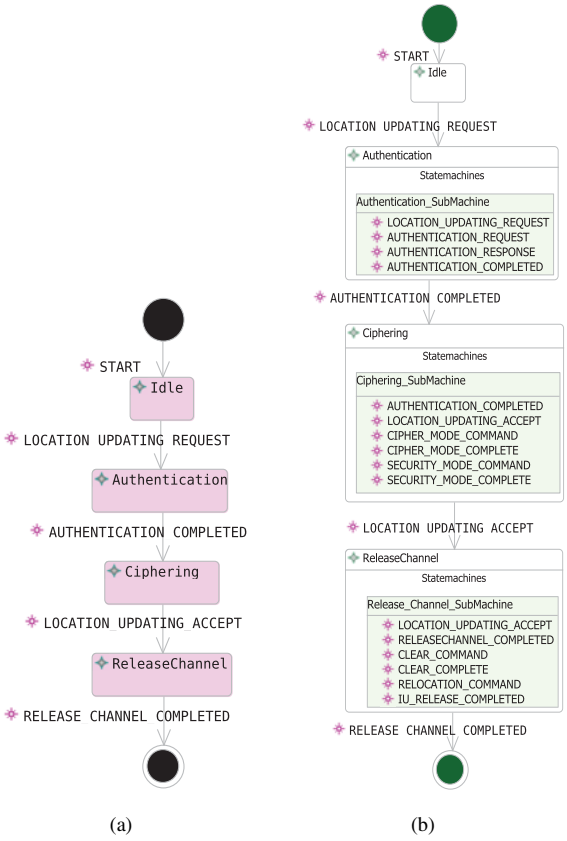


Figure 4. (a) Abstract State Machine (b) Refined State Machine

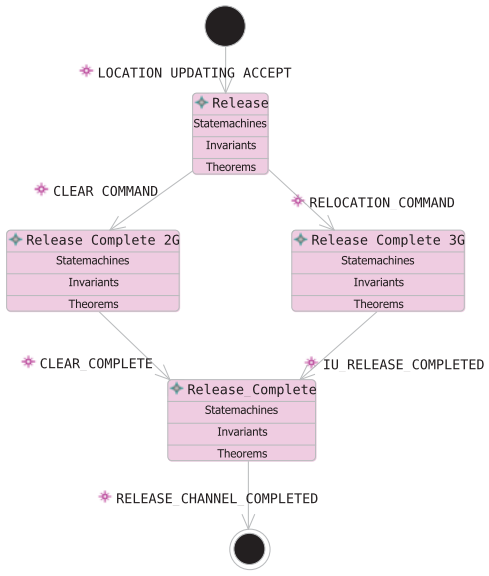


Figure 5. Release Channel Sub-state Machine

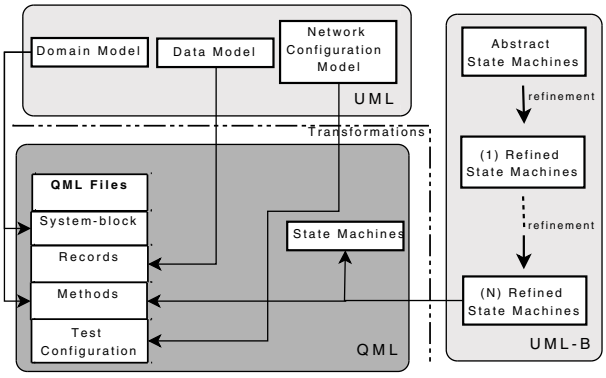


Figure 6. Transformation from UML and UML-B to QML

In UML-B, a transition is implemented as an Event-B event. The structure of events in Event-B has already been described in Section 2.3.2. An event consists of two parts, one for the guards and other for the actions. There are no external triggers in Event-B, therefore in order to represent an external trigger, e.g, reception of a message, a boolean variable, as a flag, is used in conjunction with the guard of the event. The action related to sending of a message is also modeled by using a boolean variable, whose true value marks sending of the message.

In order to facilitate automatic transformation from Event-B to the corresponding constructs in QML, we annotate the labels of Event-B statements. The labels for the conditions corresponding to triggers are prepended with the string "Trigger", while the ones that correspond to the guards are prepended with the string "Guard". There are generally two types of actions, which corresponds to sending of a message and changing values of the variables respectively. The label corresponding to the Event-B action for the first type, i.e. sending of a message, is prepended with string "Send". Similarly, the actions for variable assignments have their labels prepended by "Assign" strings. Similarly, the "CurrentState" and "NextState" labels mark the current state and next state respectively. Figure 7(a) shows an excerpt of an Event-B event with labels, guards and actions.

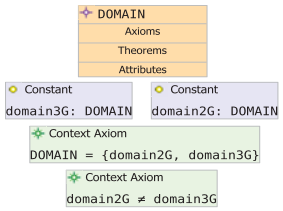
The data types in QML and UML are different than the ones supported by Event-B. In order to reduce complexity, and to concentrate on verification aspects, we abstract out data types of the variables and message parameters. For instance, in order to model a numeric type, we use `NAT` (set of natural numbers) in Event-B. Similarly, there is no *String* type in Event-B. This is handled by declaring a user-defined type and its instances (constants) as shown in Figure 7(b). For concrete test cases to be executed on SUT, a conversion between abstract and concrete data types is performed by the transformation program.

Figure 8 shows the QML sub-state machine transformed from UML-B *ReleaseChannel* sub-state machine presented

```
RELOCATION_COMMAND  ≙
STATUS
    ordinary
ANY
    self //contextual instance of class MSS
WHERE
    self.type:
        self ∈ MSS
    CurrentState.Release_Channel_SubMachine_isin_Release:
        Release_Channel_SubMachine(self) = Release
    Guard.RELOCATION_COMMAND.Guard1:
        domain = domain3G
    Trigger.RELOCATION_REQUEST_MESSAGE:
        RANAP_Relocation_Request_Message = TRUE
    ……
    ……
THEN

    Send.RELOCATION_COMMAND_MESSAGE:
        RANAP_Relocation_Command_Message := TRUE
    NextState.Release_Channel_SubMachine_enterState_Release_Complete
    _3G:
        Release_Channel_SubMachine(self) := Release_Complete_3G
    ……
    ……
```

(a)



(b)

Figure 7. (a) Excerpt of an Event-B event (b) User defined type

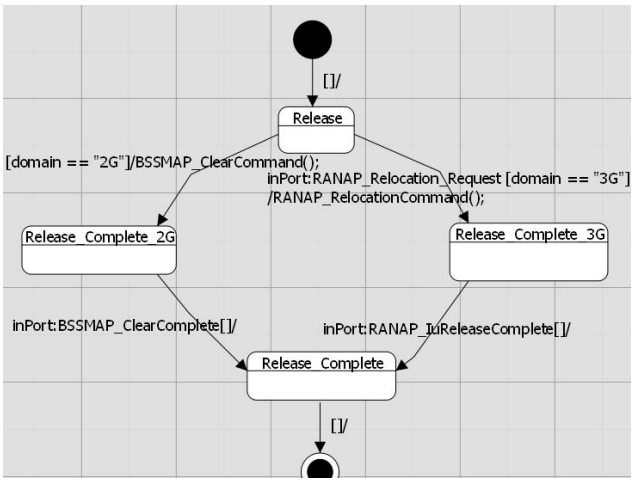


Figure 8. Release Channel Sub-state Machine in QML

in Figure 5. The messages sent or received by the system are modeled as records in QML. All the messages sent or received by the system are defined in the *system* block with their respective *Inbound* or *Outbound* ports as shown in Listing 1. The QML implementation of sending the "RANAP_Relocation_Command" message, from Figure 7(a), is depicted in Listing 2. From this implementation, it can be observed that first an instance of message/record type is instantiated and then sent to the outbound port.

Tool Support. The UML-B tool is used as a plug-in to the RODIN [10] platform. In RODIN platform, the UML-B and Event-B models are internally represented as XML metadata interchange (XMI) format. The transformation from UML-B to QML is mostly performed at the XMI-level with the exception of few complex constructs which are transformed programmatically.

Listing 1. System Block: Messages and Ports in QML

```
system {
    // *** PORTS ***
    // ***Inbound Port***
    Inbound inPort:
    MM_IdentityResponse,
    MM_TmsiReallocationComplete,
    MM_AuthenticationResponse,
    ....
    RANAP_IuReleaseComplete,
    RANAP_RelocationRequestAcknowledge,
    RANAP_RelocationComplete,
    RANAP_RelocationRequired,
    RANAP_RelocationDetect,
    ....

    // ***Outbound Port***
    Outbound outPort:
    MM_IdentityRequest,
    MM_AuthenticationRequest,
    MM_LocationUpdateAccept,
    ....
    RANAP_SecurityModeCommand,
    RANAP_IuReleaseCommand,
    RANAP_RelocationRequest,
    RANAP_RelocationCommand,
    ....
}
```

Listing 2. Sending of a message in QML

```
\\**METHODS**
class Main extends StateMachine
{
  ....
  void RANAP_RelocationCommand()
  {
      RANAP_RelocationCommand ranap_relocationcommand;
      outPort.send(ranap_relocationcommand);
      return;
  }
  ...
}
```

4. Test Configuration and Execution

In order to model the test configuration, we use the UML Object diagram (see Figure 9) instantiated from the domain model specified earlier in Figure 1. The `:MSS` instance represents the SUT while the other instances represent

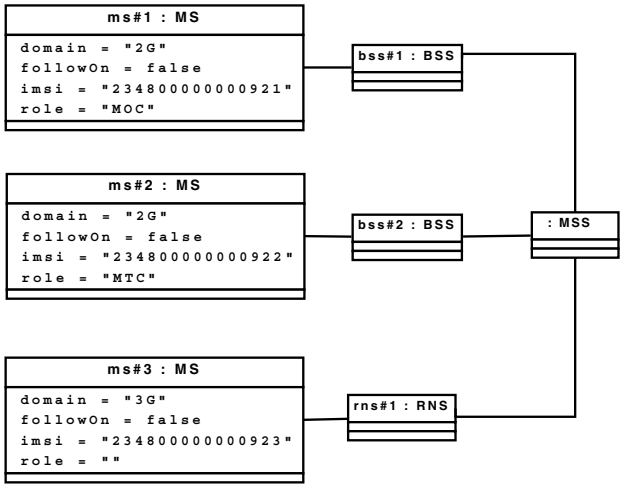


Figure 9. Test setup configuration example.

the test environment. For example, the instances of `:MS` represent different mobile subscribers requesting the location update from `:MSS`.

The information from the object diagram is used for generating the test configuration in QML. In the given example, in Figure 9, the following QML code (see Listing 3 is generated. It can be observed that subscribers are modeled as threads which are then started to test behavior of the SUT.

Listing 3. Test setup in QML

```
// *** MAIN ***
void main(){
    ...
    Subscribers mySubscribers[] = new Subscribers[2];
    ...

    mySubscribers[0].my_name = "ms1";
    mySubscribers[0].domain = "2G";
    mySubscribers[0].followOn = false;
    mySubscribers[0].registered = true;
    mySubscribers[0].role = "MOC";
    mySubscribers[0].imsi = "234800000000921";

    mySubscribers[1].my_name = "ms2";
    mySubscribers[1].domain = "2G";
    mySubscribers[1].followOn = false;
    mySubscribers[1].registered = true;
    mySubscribers[1].role = "MTC";
    mySubscribers[1].imsi = "234800000000922";

    mySubscribers[2].my_name = "ms3";
    mySubscribers[2].domain = "3G";
    mySubscribers[2].followOn = false;
    mySubscribers[2].registered = true;
    mySubscribers[2].role = "";
    mySubscribers[2].imsi = "234800000000923";

    ...
    for(int i=0; i<=2; i++){
      MSS mss = new MSS( mySubscribers[i]);
      Thread t = new Thread(mss) ;
      t.start();
    }
    ...
}
```

As the behavioral specification of the SUT is quite abstract, the test cases generated by Qtronic will be on the same abstraction level. It is the task of the Test Adaptation (see Figure 3) to concretize the messages by adding additional message parameters, their default values, and for handling communication on lower levels of the protocol stack with the SUT.

5. Related Work

There are several model-based testing approaches in practice these days. Some of these approaches use formal models while other use informal models. Linking object-oriented system development to formal specifications is not a new idea. Object-Z [12] which is an object-oriented extension of the formal specification language Z, has been used for test generation (see [13], [14]), however, these approaches do not use sophisticated graphical models and industrial scale test generation tool.

Object Constraint Language (OCL) has been used for generating test cases (see [15]), however, this approach lacks formal verification of the models.

In [16], formal specifications in Java Modeling Language (JML) are combined with test-driven development. This approach is suitable for agile development method.

In [17], a conformance testing and automatic test case generation scheme for UML Statecharts (UMLSCs) is presented. The authors propose a formal conformance-testing relation for input-enabled transition systems with transitions labeled by input/output-pairs (IOLTSs). This work, like other state chart based approaches, does not consider other aspects e.g., architectural, data etc., of the SUT.

6. Conclusions

We presented an approach on combining UML modeling with formal verification in order to improve the quality of the models used for automated test derivation. While incorporating formal verification in the over-all process gave number of advantages. Firstly, the quality of the models was improved allowing us to detect inconsistence in the models that where not detected by our custom OCL validation rules. Secondly, while modeling with UML-B, we observed several ambiguities, or not well-explained details, in the specifications that might have been difficult to observe by using UML only. In a way, by using formal specifications, we better understood the functionality of the SUT. However, it also increased the complexity of the model as we had to add more details just to *prove* the system correct.

We also found that by using formal specifications we could make the testing process a bit more efficient. For example, one can derive data dependencies between message parameters from the invariants and pre-conditions in Event-B. Invariants and pre-conditions also specify the possible restrictions on the input variables (message parameters) that

may help the test generator to reduce the input space (all possible values) for these variables.

Adjusting formal verification in our case was made easy due to availability of the UML-B tool. UML-B provides automatic translation of various object-oriented constructs. Most of the Event-B specifications were automatically generated and proved. Doing all this by hand would have been a difficult task for an engineer not very familiar with proving formal specifications.

At the moment, traceability of requirements from test cases into formal models is not performed. However, this can be done by attaching textual requirements, taken from requirement model, to the generated Event-B specifications. This would certainly help to find which part of the system has passed or failed the test and constitute a topic for our future work. Another topic for future work is on improving our UML-B to QML transformation. As the UML-B is defined based on a metamodel, we plan to create a metamodel that specifies the concepts used in QML and reimplement the transformation at metamodel level by using model-driven transformational techniques.

7. Acknowledgement

Financial support from Tekes under the ITEA2 D-MINT project is greatly acknowledged.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing*. Morgan Kaufmann Publishers, 2006.

[2] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truşcan, and J. Lilius, "Transformational Support for Model-Based Testing – from UML to QML," in *Proceedings of Model Based Testing in Practice (MoTiP'09) workshop*, 2009.

[3] J.Rumbaugh, I.Jakobson, and G.Booch, "The Unified Modelling Language Reference Manual." Addison-Wesley, 1998.

[4] Conformiq, "Qtronic," http://www.conformiq.com/.

[5] J. Abbors, "Increasing the Quality of UML Models Used for Automatic Test Generation," Master's thesis, Åbo Akademi University, 2009.

[6] "Object Constraint Language," http://www.omg.org/technology/documents/formal/ocl.htm.

[7] J.-R. Abrial, *The B-Book*. Cambridge University Press, 1996.

[8] "MATERA Approach," https://research.it.abo.fi/research/embedded-systems-laboratory/projects/d-mint/matera.

[9] C. Snook and M. Butler, "UML-B: Formal modeling and design aided by UML," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 92–122, 2006.

[10] "Rigorous Open Development Environment for Complex Systems," iST FP6 STREP project, online at http://www.event-b.org/platform.html.

[11] J.-R. Abrial, "A System Development Process with Event-B and the Rodin Platform," in *ICFEM*, 2007, pp. 1–3.

[12] R. Duke and G. Rose, *Formal Object Oriented Specification Using Object-Z*. Palgrave macmillan, 2000.

[13] D. Carrington, I. Maccoll, J. Mcdonald, L. Murray, and P. Strooper, "From object-z specifications to classbench test suites," *Journal on Software Testing, Verification and Reliability*, vol. 10, 1998.

[14] J. Mcdonald, L. Murray, and P. Strooper, "Translating object-z specifications to object-oriented test oracles," in *In Proc. Asia-Pacific Software Engineering Conference and Int. Computer Science Conference*. IEEE Computer Society, 1998, pp. 414–423.

[15] A. D. Brucker and B. Wolff, "Testing distributed component based systems using uml/ocl," in *GI Jahrestagung (1)*, 2001, pp. 608–614.

[16] H. Baumeister, "Combining formal specifications with test driven development," in *XP/Agile Universe*, 2004, pp. 1–12.

[17] S. Gnesi, D. Latella, and M. Massink, "Formal test-case generation for uml statecharts," *Engineering of Complex Computer Systems, IEEE International Conference on*, vol. 0, pp. 75–84, 2004.

Paper V

Model-Based Testing using System vs. Test Models -What is the difference?

Qaisar A. Malik, Antti Jääskeläinen, Heikki Virtanen, Mika Katara, Fredrik Abbors, Dragoş Truşcan and Johan Lilius

Originally published in *In Proceedings of 17th IEEE Intl. Conference on Engineering of Computer-Based Systems (ECBS 2010)*, IEEE Computer Society, pp. 291-299, March 2010, Oxford UK.

2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Model-Based Testing using System vs. Test Models – What is the Difference?

Qaisar A. Malik*‡, Antti Jääskeläinen§, Heikki Virtanen§, Mika Katara§,
Fredrik Abbors*, Dragoş Truşcan* and Johan Lilius*‡
*Department of Information Technologies, Åbo Akademi University, Turku, Finland.
Email: {Qaisar.Malik, Fredrik.Abbors, Dragos.Truscan, Johan.Lilius}@abo.fi
‡Turku Centre for Computer Science, Turku, Finland.
§Department of Software Systems, Tampere University of Technology, Tampere, Finland.
Email: {antti.m.jaaskelainen, heikki.virtanen, mika.katara}@tut.fi

Abstract

We discuss the differences between using system models *and* test models *with respect to the model-based testing process. Although these two terms are usually used interchangeably, very little is known about the distinction between the two. System models describe internal behavior of the system under test while the test models contain the behavior from user's or environment's point of view. We describe how these two types of models are obtained and used throughout the model-based testing process and how they are related to each other. The discussion is based on our earlier experiences as well as on two case study examples from the telecommunication domain.*

1. Introduction

The main purpose of software testing is to find defects and increase the confidence in the system under test (SUT). Functional testing, the focus of this paper, is typically done by checking the input/output behavior of the system. Traditionally, elementary tests are described by *test cases*, which state what particular part of the SUT is exercised, the input used in the test, the expected output, the identification and the description of the test case etc. Usually such test cases are grouped into *test suites*.

One straightforward way to automate testing is to develop test scripts, which execute test cases one by one. Test cases are specified using a dedicated test specification language like TTCN-3 [1] or they can be coded using a testing framework such as JUnit [2], for instance. Automation of test case execution does not, however, alleviate the fundamental difficulties that exist with test case based testing. First, linear and static test cases are not capable of finding new bugs efficiently since they do not allow variance in the tested behavior. Second, design and maintenance of test suites can be very resource consuming. These problems can be resolved to a large degree if tests are derived automatically from high-level models describing the SUT behavior.

Due to model-driven software development becoming more popular, there are models defining requirements and expected system behavior. From these models, test cases can be derived directly or via some model transformations. These test cases are linear in nature and are generally represented as scripts which are used in *off-line model-based testing*.

However, linear test cases [3, page 187], consisting of a linear sequence of events, are not suited well for testing a system that is supposed to operate in a concurrent and reactive environment. Too often a test run results in an inconclusive verdict, because the SUT responds in a valid but unexpected way to a given stimulus. A linear test case does not describe multiple expected output states. To overcome this difficulty, *online model-based testing* can be used. In *online model-based testing*, the test generator has an online connection to the SUT through an adapter component and the SUT responses affect the test generation.

The inputs for the test generation are models and coverage criteria, and the outputs are the sequence of *test operations*, the test verdict and diagnostic information (e.g. a test log) that can be used for reporting and debugging purposes, among other types of analysis. The test operations can mimic the elements the traditional test cases consist of. For instance, we can call function X with parameters P, press button B, or enter text I into field F. Moreover, the SUT response can contain information on what was the return value of the last function call R, does the last function call throw exception E, or is there text T on the display, for example. From the model and test generation heuristics points of view, the test operations are atomic, but the adapter may break them into more detailed events before sending them to the SUT.

Any model type used in model-based testing has to define the expected behavior of the SUT by the means of the interfaces of the SUT. The viewpoint of modeling can be either internal or external with regards to these interfaces. Internal viewpoint means that the corresponding model is in a passive role; it describes how the SUT should respond to given stimulus on the interface. These kinds of models are

978-0-7695-4005-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ECBS.2010.41

called *system models* (SM). On the other hand, *test models* (TM) define the behavior of the SUT from an external point of view and explicitly state what events the SUT should accept at a certain moment.

In this paper, we try to find out how the difference in the viewpoints affects the modeling process and the actual testing. Basic questions are what kind of information the modeling can be based on, what kind of development models can be used to replace dedicated TMs, what can be tested, and what kind of benefits each approach provides. Toward these ends, we describe two case study examples. In the first one, model-based testing is applied to test functional features of a Mobile Switching Server. The second case study example is about model-based testing of a smartphone application. The organization of this paper is as follows. First, some background theory and concepts about model-based testing, system modeling and test modeling are briefly introduced in Section 2. In Section 3, we present the two case studies. In Section 4, we do comparative analysis of the modeling approaches used in the case studies presented in this paper. Finally, Section 5 draws some conclusions.

2. System vs. Test Models

There are several modeling languages used for test modeling, some based on the Unified Modeling Language (UML) [4], e.g., UML2 Testing Profile (U2TP) [5], essential Test Modeling Language (eTML) [6] and some on non-UML based specification languages like CSP [7] or Labelled Transition Systems (LTS) [8]. For a comprehensive list, please refer to [9, page 62]. The choice of representation and notation depends on the intended use and expertise of the modeler. Other than choice of representation and notation, it is also very important to decide what to model in order to obtain the maximum benefit from test generation.

The primary use of models in model-based testing is to automatically create abstract specifications of the tests. As already mentioned, the viewpoint of modeling can be either internal or external with regards to the interfaces of the SUT. In the former case the corresponding model is an SM. It is in a passive role and describes how the SUT should respond to given stimulus on the interface. An SM describes the partial or complete behavior of the SUT. On the other hand, TMs define the behavior of the SUT from an external point of view and explicitly state what events the SUT should accept at a certain moment. That is, in the terms of reactive systems, TMs provide stimuli and observe the SUT reactions, while the SMs expect the stimuli and provide reactions.

Development models are those system specifications used for specifying the system at initial stages of the development process and can be used for code generation. Such models are also referred to as SMs. It is important to mention that the SM used for testing is similar, even some times identical, with the one used for the initial phases of the software development process, thus more abstract than the design models used for implementing the SUT. On the other hand, TMs are created by test engineering or test modeling experts based on any existing models and documentation, or the actual SUT behavior using reverse engineering.

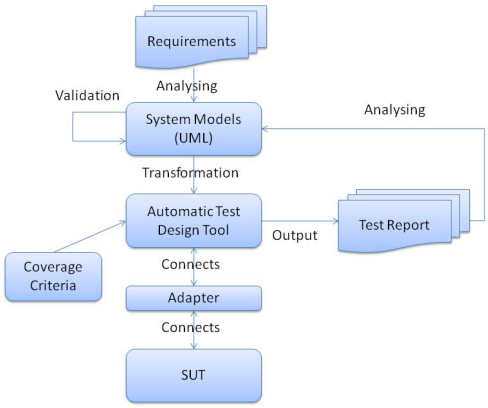


Figure 1. Testing Process: Case Study Example 1.

3. Case studies

In this section we present two model-based testing case study examples which use SMs and TMs, respectively.

3.1. Case Study 1: Testing Functional Features of a Mobile Switching Server

In the following, we will discuss an approach for using SMs for testing the system. The approach has been developed for and used in the telecommunication domain.

3.1.1. General Description. The system under test is a Mobile Switching Server (MSS). An MSS is a network entity located in a mobile telecommunication network. The MSS communicates with its surrounding elements through several different interfaces. The three main features of the MSS to be tested are location updating procedure, voice call procedure and handover. The location update procedure enables a mobile subscriber (MS) to inform its position in the network to the MSS. The voice call procedure enables the MSS to connect calls between MS's and the handover procedure enables the MSS to track the movement of MS's during an ongoing call. In the following, the examples and excerpts from case study 1 are presented for the location update procedure only.

3.1.2. Testing Process. The testing process for this case study example can be divided into the following phases as also shown in Figure 1.

Requirements. As an initial step of the approach, we create a specification starting from informal requirements (including protocol specifications, standards, user scenarios, etc.). As the SM of the SUT is derived from the informal requirements, it is important to track how different requirements reflect in the models, on different perspectives and on different abstraction levels. It is also important to propagate requirements through the test generation and execution processes, so that one can verify which parts of the models and consequently, which requirements have been tested and validated.

In our approach, requirements are first structured hierarchically (see Figure 2) using SysML requirement diagrams [10], and then they are traced to different models or parts of the models implementing them.

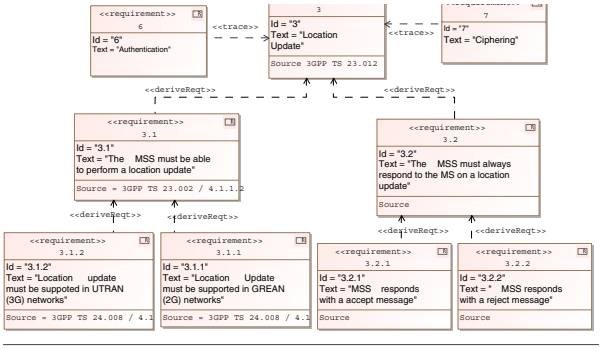


Figure 2. Requirements model example.

System modeling. For modeling, we employ a systematic approach in which the SM (in fact, a collection of models) is created in order to capture different views of the SUT. UML is used to represent different perspectives of the system like architecture, data, behavior, such that the information contained in these models can be used not only for development, but also for automated test generation using specialized test generation tools. A set of validation rules and guidelines are defined [11] for increasing the quality of the resulting models. These rules also ensure that the models are consistent with each other and moreover, that they contain the information needed in the later phases of the testing process. Tool support is provided for automatically verifying these rules.

Several types of models are used in our test process. A *domain model* (Figure 3) shows what domain components exist and how they are interfaced at different protocol levels. Each interface specifies a set for messages that can be received or sent over it. The structure of these messages and their fields are described in a *Data model* (Figure 4-(a)). A *Behavioral models* (Figure 4-(b)) describes the intended behavior of the SUT using state machines in which the messages received or sent conform to the messages specified

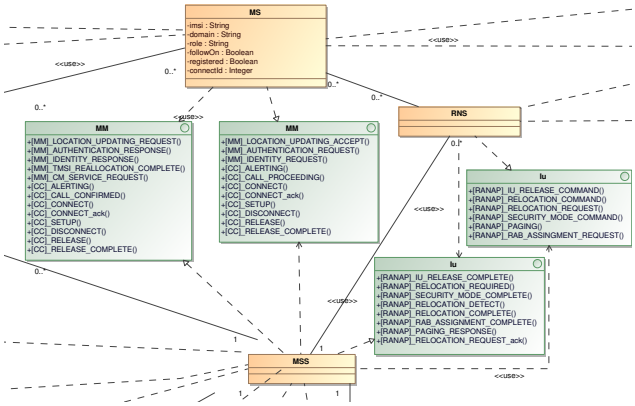
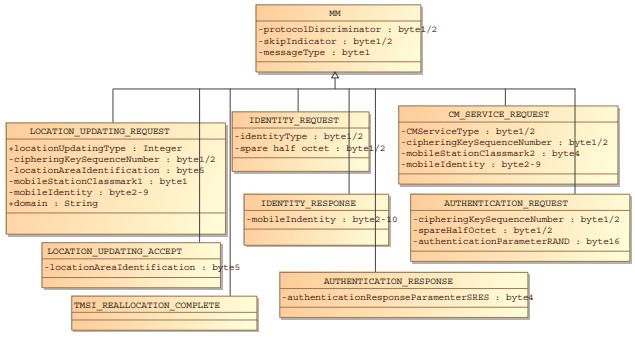
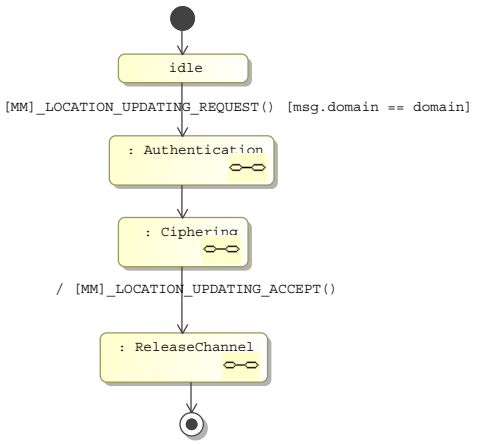


Figure 3. Domain model excerpt.



(a)



(b)

Figure 4. Data model example and Behavioral model example.

in the previous models. *Test configuration* models (Figure 5) are used to represent specific test setup configurations using object diagrams.

Test Design. The SM is used as an input for the test design phase from where test cases are obtained for either online

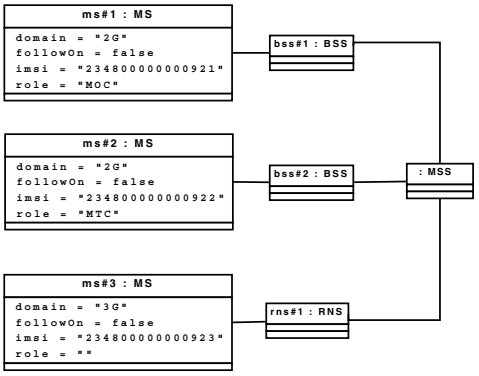


Figure 5. Test setup configuration example.

or offline testing. In this paper, we will focus on online testing only. The Conformiq Qtronic testing tool [12] is employed as the test case design tool. Qtronic accepts as input a SM of the SUT from which it automatically designs test cases according to the selected coverage criteria. The input model can be expressed as a combination of UML state machines and Java-like action language, which in our case study example, is automatically generated from the UML SMs via a model transformation [13]. For test generation, different coverage criteria types can be manually selected from the GUI of the tool like requirements, state, transitions, paths, conditional, or statement coverage. The generated test cases are sequences of input/output messages and their data values derived from the SMs to be sent/received by the SUT.

Adaptation. As the designed test cases are at the same abstraction level as the SM, an adapter is used to concretize the tests. Qtronic communicates with the adapter via messages carrying a generic data type, *datum*, which can be used to represent any concrete data type (string, integer, or composite data types). The adapter transforms the datum into concrete data types and then sends them to the SUT via the specific interface (e.g., ports, sockets). It is also the adapter that is in charge of receiving output messages from the SUT, abstracting and forwarding them to Qtronic. Qtronic will compare the received messages with the expected ones and provide a verdict for each test in part. A test report is provided with statistics about the test run regarding the number of generated test cases, pass/failed verdicts, coverage reports.

Analysis. As the Qtronic model of the SUT is automatically generated from the UML models, requirements will be propagated as well. Thus, when tests are generated and executed by Qtronic, the tool will keep track of which requirements have been covered, in which test cases, and what was the verdict of the test cases covering each requirement. At the end of the test run, we collect information from the test log about what requirements have been covered and validated by the generated set of test cases. Using requirements traceability one can also trace back the test cases to the SMs from which they where generated [14]. Such an approach proved beneficial in identifying the source of failed test cases and in debugging the models.

One well-known issue in the current testing practice is the fact that detecting the source of an error is a non-trivial problem. Typically the source of the error can be either in the SUT, in the SM, or in the adapter. In our case study example, several errors have been discovered and they originated from all three sources. For instance, some errors originated from inconsistencies discovered in the SM, which were due to misunderstanding of requirements or to incomplete validation of models before testing. Other errors have been found in the adapter, as well as in the SUT.

3.2. Case Study 2: GUI Testing of a Mobile Application

TEMA toolset [15] is targeted for GUI testing of mobile applications and has been successfully applied in finding defects from applications already on the market [16]. Modern smartphones include several applications, such as calendar, camera, and media player that resemble their desktop counterparts closely in the terms of functionality. However, due to the limitations of mobile devices in display and keyboard size, the GUIs are usually somewhat simpler.

General approach. In TEMA toolset, there are three logical parts in the test setup. The TM contains the behavior of the SUT modeled from a user perspective, the test generation heuristic selects the real actions and checks that the SUT is probed with, and the adaptation that executes actions and checks at the SUT. The test generation heuristics can be seen as a black box where the TM goes in and a sequence of test operations comes out. The operations are handed over to the adapter, which takes care of their execution. The results of execution are returned to the generation heuristics, which can use them to further guide the generation.

The TM is not created directly. What is created is a *model package*, which contains a number of *model components*. At the beginning of a test run the components are composed into the TM based on the *test configuration*, for example, how many devices are to be included in the test run and what functionality is to be tested in each. The model components fall into four logical categories: action machines, refinement machines, localization tables and data tables.

Action machines are the most important of the model components. They are behavioral models which define what is tested by describing the functionality of the SUT with *user operations*. A user operation is a function of the SUT offered to the user, such as sending a text message or playing a sound clip. They can be thought of as small-scale use cases. User operations are generic in nature and not tied to the details of the UI. This allows the action machines to

be reused for different SUTs, such as Symbian and Linux phones.

A single action machine can be seen as a process in a single processor multitasking system. A single action machine is active at a time; the others are inactive, but retain their states and can resume execution when swapped into activity. This corresponds to the behavior of applications in smartphones: one application is in active use while others remain in the background. In practice it is often necessary to use several action machines to model a single application in order to limit the size of the individual models. In such a case the models are synchronized tightly to prevent them from acting independently.

Refinement machines contain the UI implementation for the user operations in action machines. The implementation is a generally linear sequence of test operations. The result is similar to macro expansion: in model composition the user operations are divided into starting and ending phases, and the test operations inserted in between. *Localization tables* are used to perform a similar task for individual actions by replacing symbolic names with UI text strings. User operations and test operations correspond to action words and keywords described in [17], [18].

Data tables contain the data used in test generation. Unlike localization tables, this data can be structured, such as the contents of a multimedia message or full contact information. Data tables are accessed in *data statements*, which are Python code embedded into user and test operations. Data statements can also be use other Python functionality, such as the date and time libraries.

When the TM is assembled, action machines and refinement machines are combined in a variant of process algebraic *parallel composition*. The composition may be performed either all at once at the beginning of the test run, or on the fly during it. The composed model, as well as both action and refinement machines, use the LSTS (Labeled State Transition System) state machine semantics. The formalism is described in more detail in [19]. The effects of localization and data tables are factored into the executed operations later on, as they are selected by the test generation heuristics.

The events of a test run are recorded in a log, which can be used to repeat the test run or debug it. Since the log contains the executed action words, it can also be used to map the test run back to requirements.

Test modeling process. Concerning the relation of TEMA models to other artifacts, they model the end user behavior, i.e., what can be done through the GUI. Thus, they correspond to high level GUI specification on the behavioral part. However, they do not contain any information other than what is needed for testing through a GUI; implementations details, GUI widgets etc. are not referred to.

Figure 6 illustrates the TEMA test modeling process. The

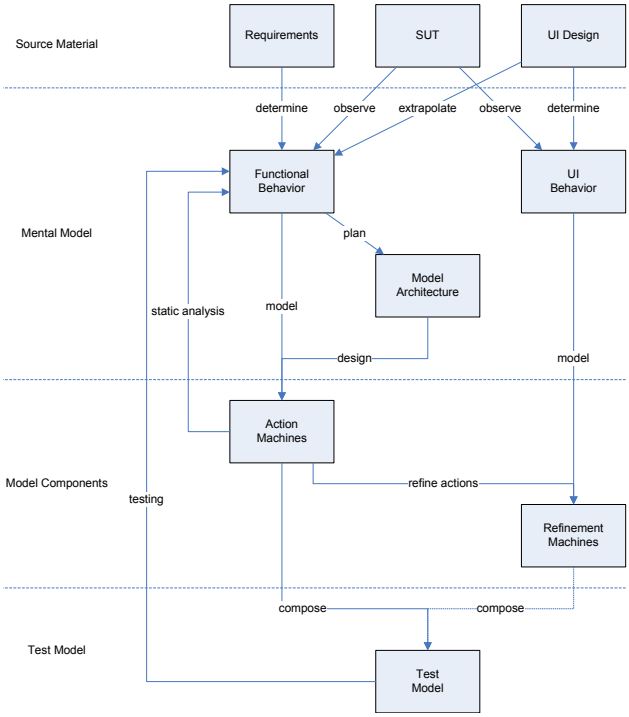


Figure 6. The TEMA test modeling process.

source material for the models may be a list of requirements, a UI design document or model, a working SUT, or some combination. From these, the modeler creates a mental model of the functionality to be modeled, and develops a plan for the model architecture, that is, what models will be created and how they interact. Based on the architecture, the functionality to be modeled is incorporated into action machines. Working with the action machines and their interactions may help the modeler to further develop his mental model of functionality, and might even uncover bugs and other issues. If information on the UI behavior is available, the modeler can then create refinement machines to implement the functionality of the action machines. Finally, the action machines and refinement machines (if available) will be combined into a single TM, which can be used for testing. A usable TM can be created without refinement machines, but tests generated from such a model would be abstract and not automatically executable.

Most of the TEMA model library has been created using reverse engineering, i.e. using the applications running on the actual device. However, we have also made one experiment where a certain application was first modeled using reverse engineering and then remodeled using a detailed GUI specification after a major revision to the user interface. The results indicated that there is no difference in the modeling effort. Obviously, the latter is more desirable way to create models, provided that detailed enough GUI specifications are

available while the functionality is still being implemented.

Case study example. As a case study example, we will study the models of the Messaging application. The application is a good representative, since it contains many different kinds of functionalities. Furthermore, it deals with the connections between multiple phones, and can thus be used to illustrate our methods for testing a number of systems in conjunction. Messaging is large enough that representing it with a single model would be highly impractical. We will thus have several model components which together represent the whole application.

The core of the Messaging models are four action machines corresponding to the most significant views of the application: Main, Inbox, SMS (short messages) and MMS (multi-media messages). These models are mostly concerned with control-related aspects of Messaging, such as moving between the views. The SMS and MMS models handle the creation of messages, although not sending them. Another model similar to these four is Startup, which is responsible for launching and exiting the application.

The Inbox model performs tasks related to the messages within the inbox, such as opening or deleting them. Since these tasks depend on the existence of messages, we will add two models to keep track of them: Messages and Messages Interface. The Messages model is essentially a variable for keeping count of the messages. Since a state machine cannot hold an infinite count, the number is abstracted into three choices: messages exist, no messages exist, and unknown. Messages Interface, as the name implies, acts as an interface for Messages, simplifying its use.

The sending and reception of messages are tasks complex enough to warrant their own models: Sender (Figure 7) and Receiver. The main complication is that the test configuration may include multiple phones, which means multiple potential Receivers. The primary task of the Sender model is therefore to activate the correct Receiver model to accept the message.

For all these action machines we need corresponding refinement machines (Figure 8) to implement their actions. Refinement machines are generally relatively simple to design, since the only thing necessary is to define the keyword sequences for the action words. For greater flexibility, the refinement machines do not include GUI text strings, but only abstract names referring to them. The actual text is defined in localization tables (Figure 9).

The SMS and MMS models obtain the contents of the messages they create from data tables. For example, the Multimedia Messages data table (Figure 10) contains items that define the subject and text of a message, as well as the types and names of attachments.

With these model components we can perform the model composition to obtain the executable TM. During the composition we define the actual devices for which the model

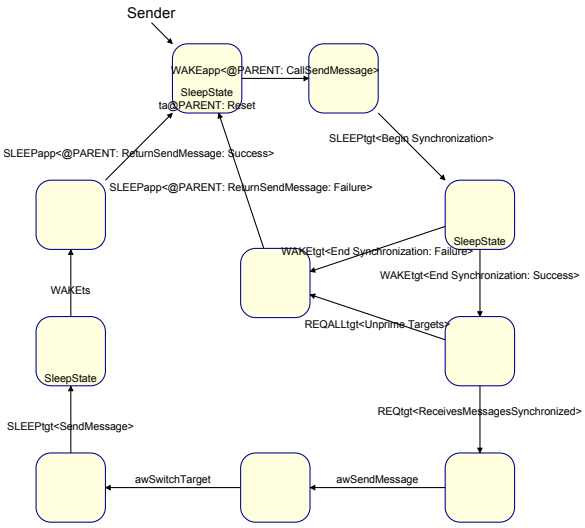


Figure 7. Example of an action machine: the Sender model.

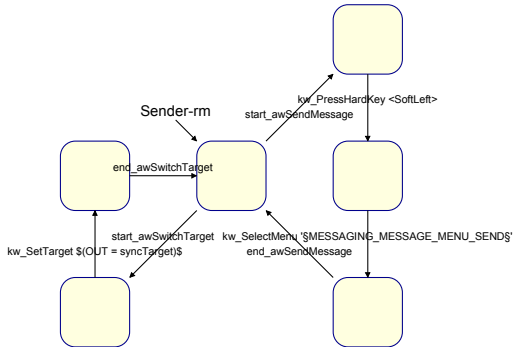


Figure 8. Example of a refinement machine: the Sender-rm model.

Messaging	en
MESSAGING_MESSAGE_MENU_SEND	Send
MESSAGING_MMS	Multimedia
MESSAGING_MMS_MENU_IMAGE	Image
MESSAGING_MMS_MENU_INSERTEXISTING	Insert object
MESSAGING_MMS_MENU_SOUNDCLIP	Sound clip
MESSAGING_MMS_MENU_VIDEOCLIP	Video clip

Figure 9. Part of the Messaging localization table.

```
multimediamessages(subject,text,objects):
[('Multimedia Message', 'Multimedia message with an image.',
[('IMAGE','multimedia')]),
('Multimedia Message', 'Multimedia message with a sound clip.',
[('SOUNDCLIP','multimedia')]),
('Multimedia Message', 'Multimedia message with a video clip.',
[('VIDEOCLIP','multimedia')]),
('Multimedia Message', 'Multimedia message with everything.',
[('IMAGE','multimedia'),('SOUNDCLIP','multimedia'),
('VIDEOCLIP','multimedia')])]
```

Figure 10. Example of a data table: MMS messages.

is intended, for example certain two phones which we want

to test sending messages to each other. Copies of all the necessary components are made for each included device.

The composition process also automatically creates some new model components which are composed along with the manually created ones. The most important of these is the Task Switcher, which manages the active model within a single device. A similar model is the Target Switcher, which manages the active device. The last one is the Synchronizer, which is used by models such as Sender to perform the synchronization between models based on data.

Figure 11 shows a highly abstracted view of the composed

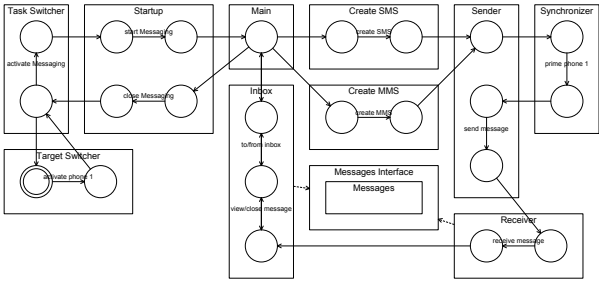


Figure 11. An abstracted view of the composed Messaging model for a single phone SUT.

Messaging model for a single phone, including the automatically generated model components. The connections between the model components correspond to the executable model, but the control flows within individual components have been shortened and simplified. The internal structure of Messages and Messages Interface is not shown at all. A TM for multiple applications or phones would be considerably more complicated.

4. Analysis

In this section, we analyze the practical differences of using SMs or TMs for model-based testing. Figure 12 summarizes the model-based testing process used in both case studies. As it can be observed, the case study example 1, shown on left hand side, uses SMs and test setup information to generate TMs. In the second case study example, shown on the right hand side of Figure 12, TMs are created by parallel composition of models.

Viewpoint of Modeling. As discussed in the beginning of this paper, one difference between SM and TM is in the way the expected behavior of the SUT is specified with respect to its interfaces; SM provides an internal viewpoint, whereas the TM provides an external viewpoint of the SUT. In fact, either of the two model types can be seen as a mirrored version of the other (TM inputs are similar to the SM outputs.)

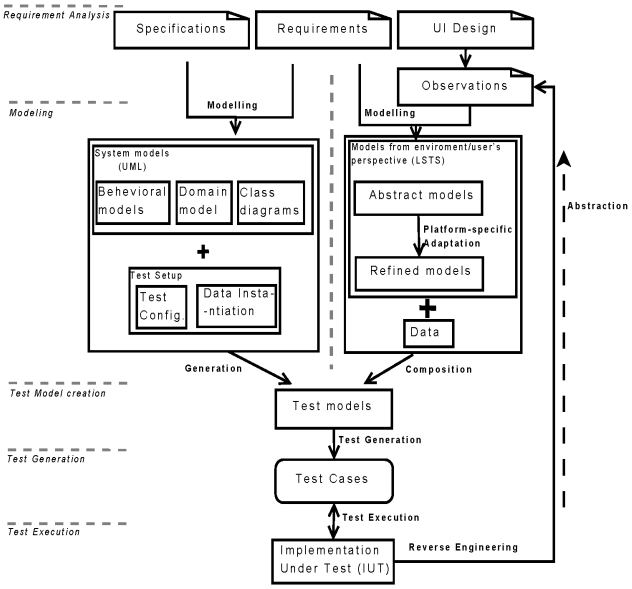


Figure 12. System Models and Test Models.

Purpose of Modeling. Concerning the purpose of modeling, the main and most obvious difference between SMs and TMs is the reason for which they are developed in the first place. If the TMs are developed solely for testing, SMs can be primarily developed for system development (however, SMs are simpler and more abstract than implementation models) and then used for testing as well. Thus, the SMs can be (partly) reused between different phases of the software life cycle, namely between development/implementation and testing, whereas TMs are only used for the latter. Both types of models typically omit some details and are in this sense incomplete descriptions of the SUT.

Requirements. It is usually considered that tests provide a ”second opinion” on the requirements: if both the implementation and the tests have been derived from the same source, i.e. the informal requirements, by developers and testers, respectively, tests not only test the implementation but can effectively reveal problems also in the requirements and in the way they have been interpreted by the developers. On the other hand, SMs are typically derived only once and then used for both implementation and testing purposes. Moreover, if the same models are used for automatic code and test generation, the tests can only reveal problems in the design and code generation phases of the development process and in the code generator.

Model Construction and Maintenance. Both types of models can either be obtained in a *top-down* approach, following the traditional software development process, or *bottom-up*

following a reverse engineering approach as depicted in the left side and the right side of Figure 12, respectively. While this is an orthogonal issue to whether use TMs or SMs, it has some practical implications. Reverse engineering obviously requires that the SUT is already mature enough. While this may seem as a significant limitation of the bottom-up approach from the practical perspective, model-based testing, as any test automation at the system level, requires the SUT to be in any case mature enough for automatic test execution, limiting its use in the early phases of the development. However, modeling can typically reveal even more defects than test execution, and it does not require the SUT in the top-down approach. While this approach benefits from being able to produce models before the SUT is even partially implemented, the models may need considerable maintenance when changes occur in the requirements during the implementation phase. However, maintenance is also needed in the case of reverse-engineering, when the version of the SUT changes, for instance.

Modeling Effort. Based on our experience, there are no significant differences in the effort needed to create the models, either SM or TM. In both cases, it is worthwhile to do static checks on the models to find inconsistencies; in SM case, this can reveal problems also in the actual requirements already before the implementation is started. However, in SM approach, any specification errors in models that are propagated to SUT are difficult to identify. On the other hand, in the reverse-engineering based approaches, both for SM and TM, it is easy to miss requirements that have not been implemented. It is also possible to model irrelevant implementation details, thus good abstraction skills are needed. Moreover, encoding the test oracle in the models requires additional information. If no specifications are available, such information can be provided by heuristic consistency [20] (consistent with user expectations, purpose, etc.).

Modeling Notations. According to our observation, TMs and SMs are not dependent on the modeling notations used. For instance, in the second case study example (Section 3.2), the TMs are modeled using LSTS (Labeled State Transition System) which can be used to model SMs, too. Similarly, TMs can be modeled using UML models. However, in practice UML models can be seen as more suitable for modeling SMs as UML has rich set of diagrams for representing internals of a system.

Black-box or White-box. The TMs are used to specify interactions with the SUT from an external perspective. Hence, it can be concluded that TMs are more suitable for black-box testing. On the other hand, the SMs specify internals of the SUT, in detail or abstractly, using some graphical notation or code. Now depending on the abstraction level of SMs, one can conclude that SMs are more suitable for white-box, or less strictly speaking, for gray-box testing.

Coverage Criteria. The two most commonly used coverage criteria are code coverage (the statements, paths, or decisions) and requirement coverage. In TM based approaches, implementation is seen as a black-box thus it is hard to give any verdict about how much of the implementation code has been covered by generated test cases unless source code is instrumented for this purpose. Therefore, requirement coverage is mostly used in this case. On the other hand, in SM based approaches, both code and requirement coverage can be observed, again provided that the implementation code is available for such analysis.

Fault-detection. It can be concluded that both TM and SM based approaches are equally effective in finding bugs. It can not be concluded that one is superior to the other in fault detection. It also depends on the details that models represent. Whereas, the quality of models is also another factor.

5. Conclusions and Future work

In this paper, we have compared two approaches to model-based testing, one using SMs and the other using TMs. The former are developed from the perspective of the implementation, while the latter sees the implementation as a black-box. In the terms of reactive systems, TMs provide stimuli and observe the SUT reactions, while the SMs expect the stimuli and provide reactions.

Even though some may claim that the use of SMs and TMs differ significantly, our conclusion is that the difference between the two lies somewhat in the eye of the beholder. However, first considering the overall software development process, if there are some development models that can be used for test generation (state machines etc.), it should be easier to transform those automatically or semi-automatically to SMs than to TMs. On the other hand, the differences may be domain-specific; in GUI testing, for instance, TMs describing the user behavior seem much more natural modeling paradigm than using SMs. Second, top-down test modeling, as traditional test case development, provides a ”second opinion” on the requirements and may thus reveal requirements based issues effectively.

Model-based testing has been in industry and academia for more than a decade and today several commercial and academic tools exist. However, we have not found any related work comparing difference between using SM and TM for model-based testing. There have been few comparisons in the literature (see [21] and [22] for example) between different model-based testing tools and techniques but these do not relate to our work presented in this paper.

In our future work we plan to elaborate our analysis by taking more case studies into account and extending the analysis to other application domains. We also hope, with this paper, to spark the interest of the software testing community on this issue.

6. Acknowledgement

Tampere University of Technology gratefully acknowledges partial funding from Tekes, Nokia, Ixonos, Symbio, Cybercom Plenware, F-Secure, Qentinel, Prove Expertise, as well as the Academy of Finland (grant number 121012). Åbo Akademi University acknowledges financial support from Tekes under the ITEA2 D-MINT project.

References

[1] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation TTCN-3," *Comput. Netw.*, vol. 42, no. 3, pp. 375–403, 2003.

[2] "JUnit 4," http://www.junit.org.

[3] D. Graham, E. V. Veenendaal, I. Evans, and R. Black, *Foundations of Software Testing ISTQB Certification*. Cengage Learning, 2008.

[4] "Unified Modeling Language 2," http://www.uml.org/, April 2009.

[5] P. Baker, Z. R. Dai, J. Grabowski, . Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker and C. Williams, "UML 2 Testing Profile," conquest 2004, ASQF Press, September 2004, Nuremberg, Germany.

[6] M. Busch, Z. R. Dai, R. Chaparadza, A. Hoffmann, L. Lacmene, T. Ngwangwen, G. C. Ndem, H.Ogawa, D. Serbanescu, I. Schieferdecker and J. Zander-Nowicka, "Model Transformer for Test Generation from Test Models," conquest 2006, September 2006.

[7] S. Nogueira, A. Sampaio, and A. Mota, "Guided test generation from csp models," in *ICTAC*, 2008, pp. 258–273.

[8] J. Tretmans, "Conformance testing with labelled transition systems: implementation relations and test generation," *Comput. Netw. ISDN Syst.*, vol. 29, no. 1, pp. 49–79, 1996.

[9] M. Utting and B. Legeard, *Practical Model-Based Testing: A tools approach*. Morgan Kaufmann Publishers, 2006.

[10] "SysML (Systems Modeling Language)," http://www.sysml.org/, April 2009.

[11] J. Abbors, "Increasing the Quality of UML Models Used for Automatic Test Generation," Master's thesis, Åbo Akademi University, 2009.

[12] "Conformiq Qtronic," http://www.conformiq.com/.

[13] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truşcan, and J. Lilius, "Transformational Support for Model-Based Testing – from UML to QML," in *Proceedings of Model Based Testing in Practice (MoTiP'09) workshop*, 2009.

[14] F. Abbors, D. Truşcan, and J. Lilius, "Tracing Requirements in a Model-Based Testing Approach," in *Proceedings of The First International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*, 2009.

[15] A. Jääskeläinen, M. Katara, A. Kervinen, H. Heiskanen, M. Maunumaa, and T. Pääkkönen, "Model-based testing service on the web," in *Proc. TESTCOM/FATES 2008*, ser. Lecture Notes in Computer Science. Springer, Jun. 2008, no. 5047, pp. 38–53.

[16] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen, "Automatic GUI test generation for smartphone applications - an evaluation," in *Proc. Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE CS, May 2009, pp. 112–122 (companion volume).

[17] M. Fewster and D. Graham, *Software Test Automation: Effective use of test execution tools*. Addison–Wesley, 1999.

[18] H. Buwalda, "Action figures," STQE Magazine, March/April 2003, pp. 42–47.

[19] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara, "Model-based testing through a GUI," in *Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005)*, ser. Lecture Notes in Computer Science, no. 3997. Springer, 2006, pp. 16–31.

[20] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley, 2001.

[21] J. M. Clarke, "Automated test generation from a behavioral model," in *Proceedings of the Eleventh International Software Quality Week*, 1998.

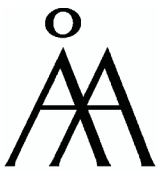
[22] A. Hartman, "Model based test generation tools," http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf. Accessed October 2009.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2467-6
ISSN 1239-1883