Fredrik Abbors

Model-Based Testing of Software Systems

Functionality and Performance

Turku Centre *for* Computer Science

TUCS Dissertations
No 202, September 2015

Model-Based Testing of Software Systems

Functionality and Performance

Fredrik Abbors

To be presented, with the permission of the the Faculty of Science and Engineering at Åbo Akademi University, for public criticism in Auditorium Gamma on September 4, 2015, at 12 noon.

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

2015

Supervisors

Professor Ivan Porres
Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A
Finland

Adjunct Professor, DSc, Dragos Truscan
Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A
Finland

Reviewers

Prof. Jüri Vain
Department of Computer Science
Tallinn University of Technology
Akadeemia tee 15 A, ICT 419, 12618, Tallinn
Country Estonia

Prof. Detlef Streitferdt
Department of Computer Science and Automation
Ilmenau University of Techology
Zuse-Building, Room 3046, 98693 Ilmenau
Country Germany

Opponent

Prof. Jüri Vain
Department of Computer Science
Tallinn University of Technology
Akadeemia tee 15 A, ICT 419, 12618, Tallinn
Country Estonia

ISBN 978-952-12-3247-3
ISSN 1239-1883

It is often said that being a Ph.D. is like being an expert on a specific topic. But are you really an expert? What do you really know? What does it even mean to know? For me, it is not so much about the knowledge, as it is about the process of learning. In order to know, you have to constantly test yourself. Ask yourself questions for which you do not know the answer. Make a prediction based on your current knowledge and your beliefs. Then collect the data and see if it matches your prediction. Anytime you come across a new piece of data, ask yourself whether it matches what you would have predicted.

If these answers do not quite match your predictions, do not just rationalize it or sweep it under a rug. Be honest. Admit that your beliefs have been incorrect or imprecise. Alter them to fit the new data, in a manner that still makes overall coherent sense. Be willing to throw out your beliefs like yesterday's news paper and start from scratch, if the situation calls for it. Pick explanations that are as simple as possible, while still being complex enough to explain what you are seeing. And once you are done with all that, gather even more data, and test yourself all over again. It is a never ending loop, but with every lap that you take, you will find yourself a wiser person for it.

Abstract

Software is a key component in many of our devices and products that we use every day. Most customers demand not only that their devices should function as expected but also that the software should be of high quality, reliable, fault tolerant, efficient, etc. In short, it is not enough that a calculator gives the correct result of a calculation, we want the result instantly, in the right form, with minimal use of battery, etc. One of the key aspects for succeeding in today's industry is delivering high quality. In most software development projects, high-quality software is achieved by rigorous testing and good quality assurance practices. However, today, customers are asking for these high quality software products at an ever-increasing pace. This leaves the companies with less time for development.

Software testing is an expensive activity, because it requires much manual work. Testing, debugging, and verification are estimated to consume 50 to 75 per cent of the total development cost of complex software projects. Further, the most expensive software defects are those which have to be fixed after the product is released. One of the main challenges in software development is reducing the associated cost and time of software testing without sacrificing the quality of the developed software.

It is often not enough to only demonstrate that a piece of software is functioning correctly. Usually, many other aspects of the software, such as performance, security, scalability, usability, etc., need also to be verified. Testing these aspects of the software is traditionally referred to as non-functional testing. One of the major challenges with non-functional testing is that it is usually carried out at the end of the software development process when most of the functionality is implemented. This is due to the fact that non-functional aspects, such as performance or security, apply to the software as a whole.

In this thesis, we study the use of model-based testing. We present approaches to automatically generate tests from behavioral models for solving some of these challenges. We show that model-based testing is not only applicable to functional testing but also to non-functional testing. In its simplest form, performance testing is performed by executing multiple test sequences at once while observing the software in terms of responsiveness

and stability, rather than the output.

The main contribution of the thesis is a coherent model-based testing approach for testing functional and performance related issues in software systems. We show how we go from system models, expressed in the Unified Modeling Language, to test cases and back to models again. The system requirements are traced throughout the entire testing process. Requirements traceability facilitates finding faults in the design and implementation of the software.

In the research field of model-based testing, many new proposed approaches suffer from poor or the lack of tool support. Therefore, the second contribution of this thesis is proper tool support for the proposed approach that is integrated with leading industry tools. We offer independent tools, tools that are integrated with other industry leading tools, and complete tool-chains when necessary.

Many model-based testing approaches proposed by the research community suffer from poor empirical validation in an industrial context. In order to demonstrate the applicability of our proposed approach, we apply our research to several systems, including industrial ones.

Sammandrag

Mjukvara spelar en central roll i många av våra elektronikapparater och produkter vi använder dagligen. De flesta förväntar sig inte bara att deras apparater ska fungera som förväntat utan också att mjukvaran ska vara pålitlig, effektiv, snabb, samt av hög kvalité. Kort sagt, det räcker inte bara med att en kalkylator ger rätt svar. Vi vill också ha svaret genast, i rätt form, med minimal användning av batteriet, och så vidare. En av de viktigaste aspekterna för att lyckas i dagens mjukvaruindustri är att leverera högteknologiska produkter samt att ha en god kvalitetssäkring. I de flesta mjukvaruutvecklingsprojekt uppnås programvara av hög kvalité genom noggranna testningsprocesser. Men efterfrågan på programvara av hög kvalité ökar ständigt. Detta innebär att företagen har allt mindre tid för utveckling och testning.

En av de vanligaste aktiviteterna inom mjukvarutestning är sökandet efter fel i programvaran som är relaterade till systems funktionalitet. Dessa defekter kan till exempel vara inkorrekt beteende, felaktig output, avsaknade av funktionalitet, etc. Denna aktivitet kallas normalt för *funktionell testning*. Funktionell testning är, i vissa avseenden, ett sätt att säkerställa att mjukvaran har all nödvändig funktionalitet om anges i dess kravspecifikation.

Alla mjukvarufel är dock inte relaterade till funktionalitet. Andra aspekter av programvaran, så som prestanda, säkerhet, skalbarhet, användbarhet, etc., måste också testas. Testning av dessa aspekter kallas traditionellt för *icke-funktionell testning*. Vissa system kan sluta fungera eller kan hindra andra användare att få tillgång till systemet enbart genom att systemet är under en tung arbetsbörda som den inte klarar av. Inom ramarna för testning av programvara måste vi också ta reda på hur ett mjukvarusystem beter sig i fråga om responsivitet och stabilitet under olika belastningsförhållanden. Processen för kontroll av dessa typer av defekter kallas normalt för prestandatestning. En av de stora utmaningarna inom prestandatestning är att det utförs i vanliga fall först i slutet av mjukvaruutvecklingsprocessen, när det mesta av funktionaliteten är implementerad. Detta beror på det faktum att prestanderelaterade aspekter vanligtvis gäller för ett program i sin helhet.

Testning av programvara är dyrt, eftersom det vanligtvis kräver mycket manuellt arbete. Hailpern och Santhanam uppskattar att felsökning, testning och verifiering ibland kan sträcka sig från 50 till 75 procent av den totala utvecklingskostnaden. De dyraste programvarufelen är de som hittas efter att produkten släpps till marknaden. Den totala kostnaden för testning av programvara kommer från många olika håll, till exempel, det manuella arbetet som går in i processen för att skapa test. En av de största utmaningarna inom mjukvaruutveckling är att minska på kostnaderna för testning av programvara utan att offra kvalitéten.

Modellbaserad testning (MBT) har, under de senaste åren, föreslagits som en lösning på många av dessa problem. Fördelen med MBT är att det bygger på automatisk generering av tester från en abstrakt modell som representerar testsystemet, snarare än att skapa varje test manuellt. Generering av tester från en modell innebär normalt att man försöker täcka modellen på flera olika sätt. Om ändringar görs i programmet, så görs motsvarande förändringar även till systemmodellen och alla tester anknytna till den ursprungliga förändringen kan enkelt genereras igen. Detta bör ställas i kontrast till att manuellt behöva identifiera och skriva om alla de test som påverkas av förändringen i programvaran. En annan fördel med MBT är att sofistikerade algoritmer kan hitta komplicerade test i en modell vilka människor skulle ha svårt att hitta. Det är därför MBT idag är en av de rekommenderade teknikerna för att testa säkerhetskritiska system.

Ett av modelleringens starka sidor inom mjukvaruutveckling är att den *skiftar fokuset från implementation till design*. Detta är fördelaktigt eftersom att flytta fokuset mot design gör att man kan *höja abstraktionsnivån* och istället fokusera på vad som specificerats. Genom att investera mer i den inledande specifikationsfasen, är det möjligt att spara tid och kostnader eftersom fel upptäcks långt tidigare. Att höja abstraktionsnivån leder i sin tur till *minskad komplexitet* vilket gör att man kan *fokusera på de relevanta delarna av ett problem i stället för att ta itu med implementationsdetaljer*. På grund av dessa fördelar har många undersökt användningen av modeller inom testning av programvara. Men med mer än 30 år av utveckling, kämpar modellbaserad testning fortfarande för att hitta sin väg in i industrin, trots alla de påvisade fördelar.

Syftet med denna avhandlingen är att utforma en strategi för att skapa modeller som används för generering av tester. I denna avhandling visar vi hur vår modellbaserad testningsstrategi kan tillämpas i en industriell miljö. Vi visar att samma grundläggande idé kan tillämpas på funktionell testning samt till prestandatestning. De metoder som presenteras i denna avhandling har verktygsstöd och är integrerade med andra verktyg bekanta för programvaruutvecklingsbranschen.

Ett av de centrala problemen vi undersker i denna avhandling är att visa *hur man skall modellera* för testgenerering och att MBT kan användas i

praktiken i en *industriell miljö*. Dessutom, i de flesta fall, anses MBT endast vara lämplig för funktionell testning. Därför undersöker vi hur principerna och fördelarna med MBT kan tillämpas på prestandatestning.

Resultaten av avhandligen kan således sammanfattas som:

1. En metod för att skapa modeller för testgenerering. Detta bidrag är uppdelat i två delar; en systematisk metod för funktionell testning och en annan för prestandatestning.

2. En metod för att öka på kvaliteten på modeller som används för test generering. Eftersom modellerna används för testgenerering kommer de genererade testerna att vara på samma kvalitetsnivå som modellerna. Genom att öka kvaliteten på modellen, kan vi också öka kvaliteten på testerna och därmed undvika onödiga misstag.

3. En metod för modellering och spårning av kravspecifikationer över en modellbaserad testprocess. Detta bidrag är uppdelad i två delar; en metod för att spåra kravspecifikationer i en funktionell MBT process och en annan metod för att spåra kravspecifikationer i en prestandabaserad MBT process.

4. En metod för generering av arbetsbörda från olika belastningsmodeller som beskrivs med PTA formalism. Vi presenterar ett notationsspråk för modellerna och förklarar PTA formalismen.

5. Implementation av nödvändiga verktygen för att stödja de tidigare nämnda resultaten

Vårt arbete är uppdelat i två delar, en funktionell- och en prestandadel. Fastän delarna är olika så förblir grunden den samma, det vill säga, vi använder modeller för både funktionell- samt prestandatestning. För funktionell testning, utvecklar vi en samling UML-modeller med start från kravspecifikationen av systemet. Modeller skapas genom att följa olika riktlinjer och omvandlas i ett senare skede till input för testgenereringsverktyg. Kravspecificationerna modelleras också och spåras genom hela processen. För prestandatestning skapas också en uppsättning modeller, antingen manuellt eller från loggdata. Modellerna är probabilistiska och beskriver hur användaren interagerar med ett system. Dessa modeller används som input för generering av arbetsbörda för olika testsystem.

Acknowledgements

It is not that I would not feel a little sentimental after reaching the point of writing the final part of my thesis. It is the realization that a long and interesting journey is soon coming to an end and that a new, and hopefully equally interesting, chapter in my life is about to begin. This thesis is not the outcome of one person's hard work, but the result of much collaboration with many people of which I feel honored and privileged to have been able to work with. Without the support and feedback from these people, I do not think this thesis would have been completed. It is certainly a pleasure, to have the opportunity to express my deepest gratitude to all of those who helped me in making this thesis become a reality.

First of all, I would like to thank my professor, Ivan Porres, for trusting in me and giving me the opportunity to pursue my Ph.D degree at the Software Engineering Laboratory. It has certainly been an honor to work with him. Secondly, I would also like to thank my supervisor, Dragos Truscan, for his constant encouragement, guidance, and critique though all these years. Dragos is an excellent supervisor, teacher, and researcher, and has to me, been a person that I can always come and talk to whenever I need motivation or inspiration. I would also like to sincerely thank professor Jüri Vain and professor Detlef Streitferdt for their time and effort to review my thesis. Their valuable feedback and comments are greatly appreciated. I am also honored and thankful to professor Jüri Vain for his kind acceptance to act as the opponent at my doctoral defence.

I would also like to thank all my coauthors for giving me the chance to collaborate with them in my research. Without you, this work would not have been possible. In particular, I would like to thank Tanwir Ahmad, Dragos Truscan, Andreas Bäcklund, Ivan Porres, Johan Lilius for collaborating on this research. I would also like to thank my coauthors from the industry, namely Risto Teittinen and Veli-Matti Aho from Nokia and Jani Koivulainen from Conformiq for collaborating with me during my research. A huge thanks also goes out to people which whom I did collaboration but did not coauthor any papers. I would specially like to thank Vinski Bräysy at Nethawk and Johan Abbors, Kim Nylund, Tuomas Pääjärvi and Erik Őstman from Åbo Akademi for laying much of the foundational work in my

research. Without your work, much of my research would not have been possible.

Furthermore, I am gracefully acknowledging the IT-Department and Graduate School of Software Engineering (SOSE) for their financial support during my PhD and for providing a great working environment. Your contribution is greatly appreciated and without it, hardly any of my research would have been possible. I am also happy and privileged to have received scholarships from Nokia Research Foundation, Hans Bangs Stiftelse, and Svensk sterbottniska Samfundet.

Many things, not directly related to my research but still needed, happen behind the scenes. The job that these people are doing is not always visible, but nevertheless important. It has allowed me to focus on my research while they take care of all kinds of practical things. I wish to thank in particular Christel Engblom,Nina Hultholm, Tove Ősteroos,Solveig Vaherkylä, and Susanne Ramstedt with all your administrative support in various matters. Also Niklas Grőnholm and Joakim Storrank for their support regards technical matter and always having their door open. A special thanks also goes out to Tomi Suovuo for helping me with getting this thesis printed.

Having a nice working environment with kind and friendly colleagues, with whom you can always come and have a little chat, has certainly helped me during my research. With this in mind, I would like to thank current and previous colleagues at the Software Engineering Lab, especially, Dragos, Ivan, Tanwir, Benjamin, Irum, Adnan, Kristian, Faezeh, Ali, Max, Jeanette, Marta, Espen, Mehdi, Niclas, Martin K, and Martin R.

Sometimes it might be helpful to get input from, and discuss with, people from outside academia or your own research group. I am thinking about all the people with whom I have shared countless interesting lunch meetings. So, I would like to extend a big thank you to all of my friends outside academia, that have had the patience of listening to my tricky research problems for hours and hours. Thank you for all of your support though these years, especially, Johan Abbors, Jonas Storholm, Kim Nylund, Andreas Bäcklund, and Ronnie Snellman. Your input has been more helpful than I think you can imagine. There are also countless other people to thank with whom I have shared many interesting discussions. I would especially like to thank, Johan, Andreas, and Stefan for always having the time to discuss various matters. Finally, I want to thank my family. I am grateful for all your love and support throughout my academic career.

List of original publications

1. *Tracing Requirements In A Model-Based Testing Approach.* Fredrik Abbors, Dragos Truscan, and Johan Lilius. Originally published 2009 First International Conference on Advances in System Testing and Validation Lifecycle. IEEE Computer Society.

2. *Including Model-Based Statistical Testing in the MATERA Approach.* Andreas Bcklund, Fredrik Abbors, and Dragos Truscan. Originally published 2010 Proceeding of 3rd Workshop on Model-based testing in practice.

3. *MATERA - An Integrated Framework for Model-Based Testing.* Fredrik Abbors, Andreas Backlund, and Dragos Truscan. Originally published 2010 Proceeding of 7th Workshop on System Testing and Validation

4. *Applying Model-Based Testing in the Telecommunications Domain.* Fredrik Abbors, Veli-Matti Aho, Jani Koivulainen, Risto Teittinen, Dragos Truscan. Originally published 2012 Model-Based Testing for Embedded Systems, Taylor and Francis Group, LLC.

5. *Model-Based Testing of Web Services Using Probabilistic Timed Automata.* Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres. Originally published 2013 Scitepress, 9th International Conference on Web information Systems and Technologies.

6. *Performance Testing in the Cloud using MBPeT.* Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres. Originally published 2013 Turku Centre for Computer Science, Developing Cloud Software.

7. *An Automated Approach for Creating Workload Model from Server Log Data.* Fredrik Abbors, Dragos Truscan, Tanwir Ahmad. Originally published 2014 Scitepress, 9th International Conference on Software Engineering and Applications.

Contents

PART I

Research Summary

Chapter 1

Introduction

Software is a key component in many of the devices and products that we use every day. Most customers demand not only that their devices should function as expected but also that the software should be of high quality, reliable, fault tolerant, efficient, etc. In short, it is not enough that a calculator gives the correct result of a calculation, we want the result instantly, in the right form, with minimal use of battery, etc. One of the key aspects for succeeding in today's industry is largely due to delivering high-end products and having good quality assurance. In most cases quality is the central aspect. In most software development projects, high-quality software is achieved by rigorous software testing processes. However, today, customers are increasingly asking for these high-quality software products. This leaves the companies with less time for development [1].

There are many different definition to what software testing is and what it is not. Hetzel wrote in 1973 that *"Testing is the process of establishing confidence that a program or system does what it is supposed to do" [Hetzel 1973].* In 1979, Myers wrote that *"Testing is the process of executing a program or system with the intent of finding errors" [Myers 1979].* There are countless other examples and definitions of software testing. However, software testing can not show that the tested program or software is free from errors and can not show that a program performs its intended goal correctly to 100 per cent certainty. In fact, the well know computer scientist E.W. Dijkstra point out already in the early 1970's that *"program testing can be used to show the presence of bugs, but never to show their absence"* [2]. In this thesis we will take the definition that software testing is *"an investigation/examination conducted on a program or system to find as many errors as possible, check that it meets the requirements, and bring the software to an acceptable level of quality"*. Software testing can also be used by a business to objectively access and understand the risk of software implementation. Most software testing techniques include the activity of

executing a program or application, using specific test data, with the intent of finding errors. Software errors include a variety of possible faults but are usually referred to as undesired behavior or incorrect output from a program or application with respect to a particular set of inputs.

One of the most common activity in software testing is looking for defects in the software that are related to systems functionality. The defects could for example be wrong behavior, erroneous output, missing functionality, etc. This activity are normally referred to as *functional testing*. In some sense, functional testing is the means of ensuring that a piece of software has all the required functionality that is specified within its functional requirements.

However, not all software defects are related to functionality. Other aspects of the software, such as performance, security, scalability, usability, etc., need also to be verified. The verification of these aspects of the software is traditionally called *non-functional testing*. Some systems may stop functioning or may prevent other users to access the system simple because the system is under a heavy workload with which it can not cope. In software testing, we also need check how a software system performs in terms of responsiveness and stability under various load conditions. The process of checking for these types of defects in normally referred to as *performance testing*. One of the major challenges with performance testing is that it is usually carried out at the end of the software development process when most of the functionality is implemented. This is due to the fact that performance aspects usually apply to the software as a whole.

Because software testing requires much manual work, it is an expensive activity. Hailpern and Santhanam estimates that debugging, testing and verification can sometimes range from 50 to 75 percent of the total development cost [3]. Further, the most expensive software defects are those which have to be fixed after the product is released. The total cost attributed to software testing originates from many different places, e.g., the manual labor that goes into the process of creating tests. One of the main challenges in software development is reducing the associated cost and time of software testing without sacrificing the quality of the developed software [4].

Another drawback is that functional testing, as well as performance testing, involves tedious manual work when creating test cases. In many large scale software projects, the testing cost is proportional to the number of tests needed to demonstrate and ensure a certain level of quality. Gauf and Dustin reports that 50 percent of the software companies allocate between 30 to 50 percent of their total development time of testing [5].

A software system typically undergoes plenty of changes during its lifetime. Whenever a piece of code is changed, a test has to be updated or created to show that the change did not break any existing functionality or introduce any new defects. Maintaining all the created tests adds additional costs due to changing requirements and changes made to the source

code. In case of performance testing, this implies that one has to be able to benchmark quickly in order to effectively check if the performance of the system is affected by the change of the code.

In recent years, *Model-Based Testing* (MBT) has been proposed as a solution to many of these problems. The advantage of MBT is that is relies on automatic generation of tests from an abstract model representing the *system under test* (SUT) rather than manual crafting. Generating tests from a model normally implies covering the model in several different ways. If changes are made to the software, the corresponding changes are also made to the system models and all tests related to the original change can simply be re-generated. This should be contrasted with having to identify and manually re-write all the tests that are affected by the change in the software. Another benefit with MBT is that sophisticated algorithms can reveal complicated tests that humans would have difficulties in finding. That is why MBT nowadays is one of the recommended techniques for testing safety-critical systems [6].

In software testing, tests can be derived from many different sources, typically, specifications, requirements, design documents, and source code. Software testing activities can be divided into 5 different levels. These levels describe *when* the testing activity is taking place but also *what* to test in relation to the traditional waterfall software development model. Figure 1.1 shows how each testing level corresponds to a particular development activity. At each level different types of testing methods and activities can be deployed. We will begin at the lowest level.

In programming, at the lowest level, we find small units of code that are called functions. The purpose of functions is to carry out very specific tasks. For example, raising one number to the power of another number or sorting elements in a list. They take one or more inputs, called parameters, perform some operations, and return an output value. Unit testing refers to the activity of testing these units or functions. In principle, testing of functions is done without knowledge about other functions or encapsulating software and can be started as soon as implementation is ready. At one level higher up we find detailed design. This phase involves the activities of combining functions from the level below into modules that carry out operations. Hence, a module is a collection of related functions. Module testing is the activity of testing modules in isolation from other modules. In other words, testing without knowledge of or influence from other modules. This activity can be stated as soon as all the functions belonging to the same module has been implemented.

Going one level further up we find the subsystems design. In this phase of software development the structure and behavior of subsystems are defined. It involves the activities of connecting the right modules together to fulfill a specific goal or use case. For example, a printing module on a com-

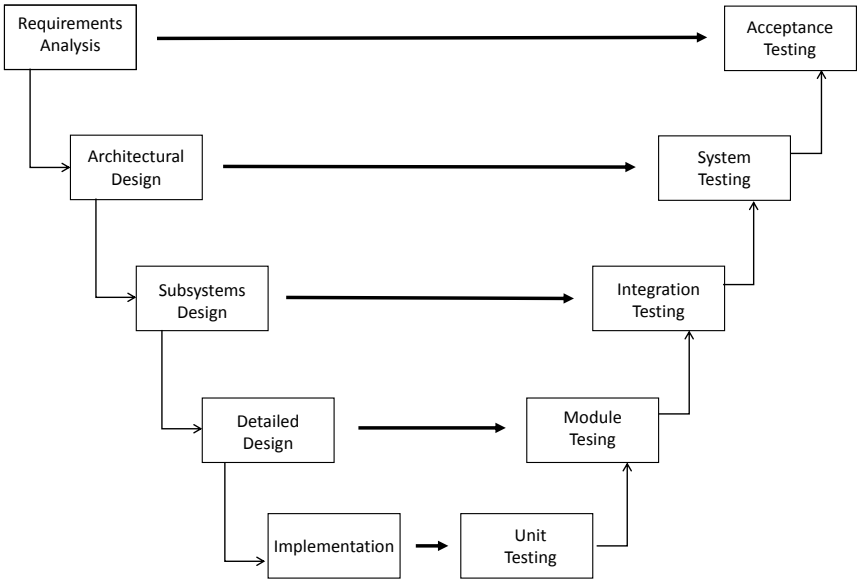


Figure 1.1: The V-model of software development and testing processes.

puter needs to be able to communicate with a module on a printer in order to print a document. Integration testing is the activity of assessing whether the interface between different modules have consistent assumptions and communicate correctly. This activity can be started as soon as two or more communicating modules have been independently tested. On the second highest level we find architectural design. This phase specifies components and connectors that together comprises the entire system whose implementation is expected to meet the system requirements. System testing is design to test the system as a whole and verify whether the requirements have been fulfilled. It assumes that all the underlying modules and functions work individually and investigates if the system works as a whole. At the highest level we have the requirements analysis phase. In this phase all the customers' needs are captured in requirements stating what the system is supposed to do. Acceptance testing is the activity of verifying if the software system actually does what the customer wants it to do. In most cases the customer is also involved in this phase.

In software testing, a clear distinction is made between *black-box testing* and *white-box testing*. *Black-box testing* is a method for designing/writing tests without looking at the internal structure or the source code of a program. The system or program is viewed as a black box without any knowledge about what is inside. Instead, this method focuses on what is externally visible or the inputs and outputs. In a sense, the tester only knows *what*

the software should to, but now *how* it does it. To write tests using this method one needs access to requirements document and specification of the software. Black-box testing can be applied to all testing levels but is typically applied only to the higher testing level (component interface testing, system testing, and acceptance testing).

On the other hand, *White-box testing* is a method for designing/writing tests when the tester has access to the program's source code. In other words, the tester also knows *how* the software produces a specific output. The knowledge of the internal working of a module or function is used for determining input values and designing tests that exercises specified and desired paths through the source code. White-box testing can be applied to all testing levels but is typically applied only to the lower testing levels (unit testing and integration testing). Figure 1.2 depicts the difference between black-box and white-box testing. For example, in the black-box case, we know only *what* the function is supposed to do but how exactly *how* it does it. In the white-box case, we have insight or knowledge about the implementation and can design test accordingly. In the context of this thesis, we focus only on black-box testing methods.

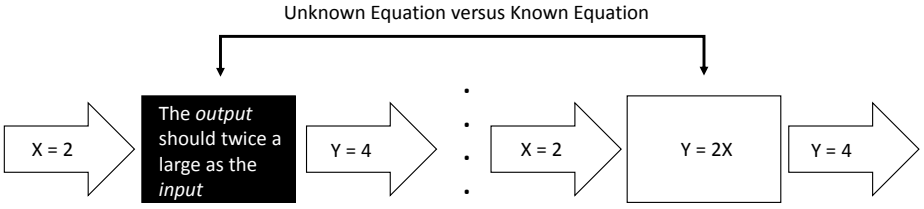


Figure 1.2: Black-box vs white-box testing.

Model-based testing (MBT) is a relative new and modern testing paradigm. It is a black-box testing technique that relies on generation on tests from abstract models rather that manual crafting. There are many definitions of MBT but Mark Utting defines it as "the generation of test cases with test oracle from a behavioral model" [7]. In other words, the system behavior is captured in an abstract model from where tests are later automatically generated. This means that the focus has been shifted form test case implementation to modeling. The test model is usually not a complete description of the system functionality but rather an abstract or partial representation of the behavior of the system under test (SUT) relevant for testing. It is up tester to decide the level of abstraction and what to represent in a model.

Ideally, the time taken to develop a model should be less than the time

taken to implement all the tests manually. However, this may vary depending on the skill of the tester. Figure 1.3 shows the difference between model-based testing and traditional testing.

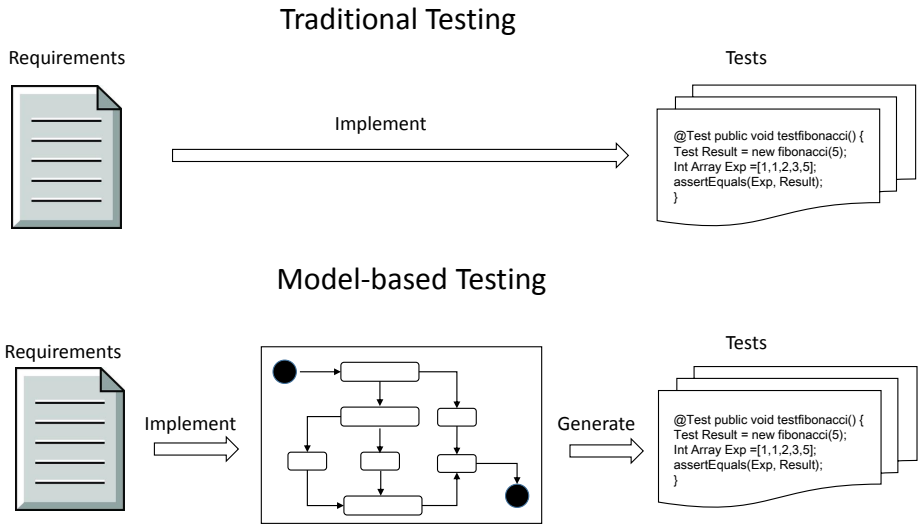


Figure 1.3: Model-based testing versus traditional testing.

One of the main advantages of MBT is test case generation. Large number of tests can be generated in a very short amount of time. Another advantage if MBT is its resilience to changing requirements. If a change is made to the system, the corresponding change can be made in test model and all the affected test can be re-generated in an instant. This should be contrasted to the traditional approach where a tester needs to locate every test affected by the change in the system and update each test accordingly. Since tests can be automatically generated in MBT, it is also believed that maintaining a model costs less than maintaining a test suite built manually [8]. Finally, the fact that modern IT system are very complex and that multiple test generation algorithms can be combined to cover a model, it is possible to uncover tests that would be difficult to discover using traditional methods.

Tests that are derived from a test model are on the same abstraction level as the test model. Abstract tests cannot directly be executed against an SUT, but need to be concretized. This a usually achieved by mapping abstract test to executable test using a form of adaptation layer in between.

Cloud computing, or simply "the cloud", is a relatively new trend in computing. In its most basic form, cloud computing is a general term for anything that offers hosted services over the internet. For instance, hardware, platforms, or services [9]. From a business perspective, cloud com-

puting focuses on maximizing the effective use of shared resources the by offering various pay-per-use models for development, deployment and scaling for software and services. From a customer point of view, cloud computing offers the possibility to access on-demand computing resources for large-scale IT infrastructures without requiring a large up-front investment. The cloud also offers the possibility to dynamically scale up and scale down the IT infrastructure depending on the capacity needed at particular times. The resources models offered by a cloud are delivered as to the customer as "as-a-service" solution and they are normally divided into three categories.

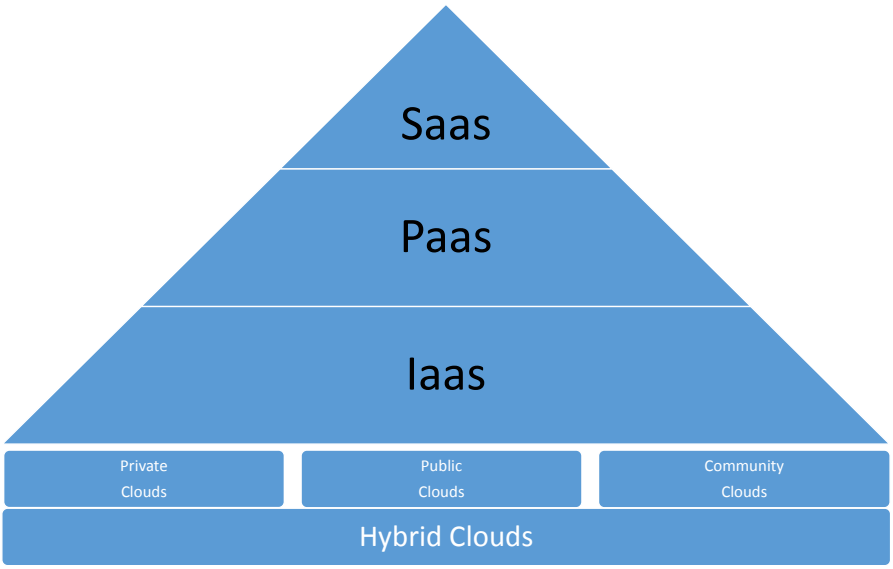


Figure 1.4: Cloud service models.

Figure 1.4 shows the tree categories. These service models can be deployed in public, private, community, or hybrid clouds environment. Different cloud providers are targeting specific customer segments by offering different combinations between service and deployment models. The Iaas service model, Infrastructure-as-a-Service, offers a complete infrastructure to its users. The infrastructure usually contains computational, storage, and network resources. The users can freely choose the operating system, applications, and services they wish to run. The PaaS service model, Platform-as-a-Service, offers its users the possibility to create, deploy, and run applications using a specific set of programming languages, tools, libraries, frameworks supported by the cloud provider. The users of Paas services control over the applications but contrary to Iaas, they do not have control over the underlying infrastructure, e.g., operating system, hardware, net-

work, etc. SaaS, Software-as-a-Service, is the most restricted service model. Here, the cloud provider allows the users to use specific applications. Limited control, such as application specific details, is given to the users.

1.1 Motivation

One of the promises of modeling in software development has been that it *shifts the focus from implementation to design* [10]. This is beneficial because shifting the focus towards design allows one to *raise the abstraction level* and think more at the beginning about what is being specified. By investing effort in the initial specification phase, it is possible to save time and effort because errors are detected far earlier [11]. In turn, raising the abstraction level *reduces complexity* and allows one to *focus on the relevant parts of a problem instead of dealing with implementation details.* Because of these benefits, many have investigated the use of models in software testing. The term "Model-Based Testing" is a relatively recent invention and was adopted sometime around the mid 1990's but its origins can be traced back to the 1970's when it was used for testing software modeled by finite-state machines [12]. With more than 30 years development, model-based testing is still struggling to find its way into mainstream industry despite all the demonstrated advantages. Bernhard et. al. [13] believe that one of the main causes is due to the *lack of proper cohesion between tools and systems used in the development and testing processes.* The area of software testing is continuously trying to improve by reducing time and cost while maintaining the same level of quality. Many believe that model-based testing could give software testing the boost it needs to meet these goals. In fact, in a model-based testing survey, the respondents report that MBT reduced the costs and testing duration with 17 percent and 25 percent, respectively, while reducing escaped bugs with 59 percent [14].

In a systematic review, Neto et. al. [15] point out that most MBT approaches have *poor integration with software development processes* and *lack empirical evaluation from industrial environments.* Many of the reviewed approaches *lack tool support* and of the 406 reviewed articles, 95 percent used a *non-UML notation language.* UML is, by many, considered the de-facto standard notation language for object oriented modeling [16],[17], [18],[19]. Almost every reviewed approach is *focusing on either functional testing or non-functional testing* without combining the two testing areas under the same approach. Many believe there is a *gap to fill in terms of research and empirical experimentation together with industry.*

This opens up new research questions on how to align current model-based testing approaches with industry testing practices and how to integrate current model-based testing tools with tools familiar to the industry.

1.2 Purpose of this Thesis

The proclaimed benefits of MBT have long been advocated by the academic world, however, the industry have not been fully committed to the use of MBT as the standard testing technique. The aim of the thesis is to design an approach for creating models used for test generation. In this thesis, we show how our model-based testing approach can be applied in an industrial setting to functional testing, as well as, to non-functional testing. We show that the same basic idea can be applied to functional testing as well as to performance testing. The approaches presented in this thesis offers tool support and are integrated with other software development tools familiar to the industry.

1.3 Research Methodology

A research methodology is a systematic process or approach used in various fields of study to support decision making or to expand a particular branch of knowledge. It is also the means of analyzing and building models to test hypothesis or to address research questions. A methodology typically contains more than simply a research method. It can constitute of modeling, analysis, testing, and other quantitative or qualitative techniques. *Design science* is a research methodology that focuses on addressing questions regarding the development of artifacts and evaluating their usefulness, performance, stability, etc., with respect to previous artifacts [20].

Figure 1.5: Design science process model.

The explicit intent is on improving upon previous artifacts. It is a commonly used research methodology in the field of information technology and differs from other methodologies typically used in natural sciences, which are problem oriented, in that design science is solutions oriented. Figure 1.5 shows the key activities in a design science approach. Once a problem has been identified, the first phase in the design science process is designing as solution for the problem that is to be solved. In the second phase the actual implementation tale place. In this phase, the actual artefact, whether it is

a model, method, or instantiation, is built. In the last phase, the produced artefact is evaluated against existing artefact to determine wether is meets its goals. The process can be iterated several times until the desired goals are met. Most of the research presented in this thesis have been in collaboration with industrial partners and have been applied on specific problems and in a specific context. We have extensively made use of design science principles because, once a problem have been identified, the design science process model best fits our work flow, i.e., designing a solution, implementing it, and evaluate.

1.4 Research Questions

According to Utting et. al., MBT consists of three parts *modeling*, *test generation*, and *test execution* [8]. For the most part, the problem of test generation and test execution has already been solved. There exist at plethora of tools and platforms for test generation [21] and test execution [22]. In the academic world, MBT has already been for many years proposed as one solution to reducing the cost and time related to software testing. However, MBT has not yet gained the momentum in the industry that many had hoped for. This might be due to the lack of sufficient tool support and tester skills required for MBT.

One of the main problems we intend to address in this thesis is to show *how to model* for test generation and that MBT in fact can be used in an *industrial setting*. Moreover, in most cases, MBT is considered to only being suitable for functional testing. Hence, we investigate how the principles and benefits of MBT can be applied to performance testing. Below are the Research Questions (RQs) that inspired this thesis.

- RQ1: How to create models that can be used for test generation?
  - Focus on the relevant things and abstract away implementation details.
  - To find out which perspectives of the system that need to be modeled.
  - Choosing the right level of abstraction.
- RQ2: How can the quality of the test models be increased?
  - Higher quality models lead to higher quality tests.
  - Ensures that all the necessary information is present in the models.
- RQ3: How to model and trace requirements across a model-based testing process?

  - Ensures that the specified system requirements are covered by tests.
  - Ensures that the specified KPIs are met.
  - Facilitates identification of which functionalities of SUT are not in sync with the model and the requirements.

- RQ4: How models for load generation can be created and used?
  - Focus on the relevant things and abstract away implementation details.
  - Ensuring that the right things are captured in the models.
  - Choosing the right level of abstraction.

- RQ5: How can tool-chain connectivity be improved?
  - Without good connectivity to industry tool-chains no one will use it.

1.5 Overview of Research Contributions

In this thesis, we investigate the use of models for functional and performance testing. Moreover, we propose new processes, methods, and tools for using models in the context of MBT. Some of the tools used were commercial while other tools were developed in-house as a proof-of-concept to fit the tool-chain.

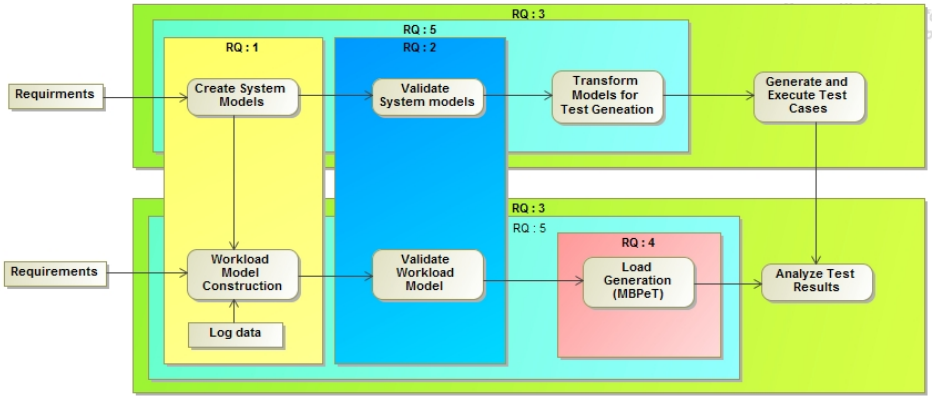


Figure 1.6: Overview of an MBT approach supporting functional and performance testing.

An overview of our MBT process is presented in figure 1.6. The figure also shows which research questions that are addressed by what parts of

the process. In the upper part of figure 1.6, we present our model-based functional testing process and show how we go from requirements to models and to tests. In the lower part of the figure, we present our model-based performance testing process and show how it is connected to the previous part.

Figure 1.7 shows a more holistic view of our process. The right side of the box show the main stages in an MBT process. On the front of the box we show our contributions for each stage in the MBT process. The rectangles with a deeper color are our contributions. On top of the box we see the how the MBT process is split into two parts, one for functional testing and one for performance testing. On the back side of the box, we have traceability of requirements which is supported throughout the whole MBT process.

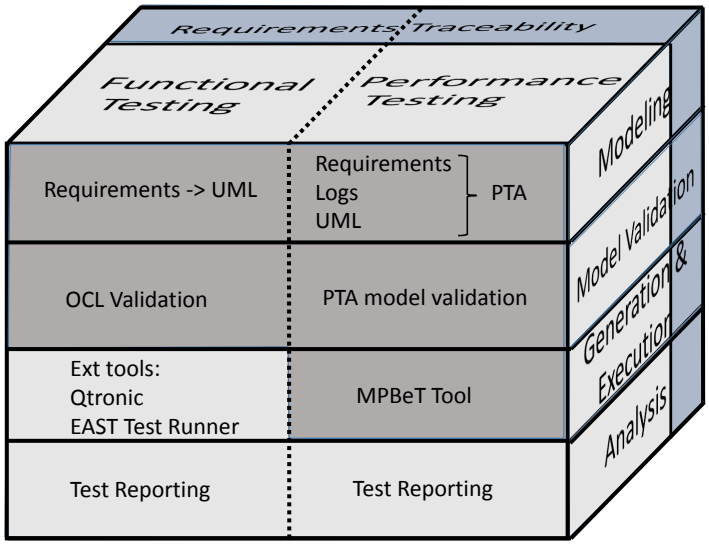


Figure 1.7: 3-dimensional view of our process.

The presented research work in this thesis has mainly been validated by conducting experiments on different example. The examples show the applicability and usefulness of the proposed approaches while the experiments give quantitative data. In the following sections we present a brief overview of the main research contributions.

1.5.1 Creating Models for Testing

Our first contribution of the thesis is an approach for creating models for testing. This contribution is divided into two parts; one systematic approach for functional testing and another for performance testing. This contribution

answers research questions **RQ 1**: How to create test models that can be used for test generation? and **RQ 4**: How models for load generation can be created and used?

In our first approach, we focus on *creating systems models used for functional test generation*. We investigate how complex systems can be models in UML, on what level of abstraction, and what perspective of the systems are needed for successful test generation. This approach is explained in *Paper III* and *Paper IV*. The systems models are created using the UML modeling language. Both static and dynamic system behavior is expressed in the models. In our publications, we present a systematic approach where models are created starting form analyzing and modeling the requirements and system features. We the express the modeled requirements in terms of use cases and sequence diagrams. From the sequence diagrams we create state diagrams and class diagrams representing the static and dynamic view of the system. Finally, we have data models depicting the different data types used in the systems and a test configuration diagram representing a specific test setup of the system under test. This work has been validated by applying our proposed approach to an industrial tele-communications case study. In the case study we modeled three features (*location update*, *phone calls*, and *call handover*) of and Mobile Switching Server (MSS).

In our second approach, we focus on the *creation of models used for load generation*. We explore the use of probabilistic models in order to describe operational profiles that are used for load generation. We present a performance testing approach for web services in which we use abstract probabilistic models to describe how users interact with the system. This research is presented in *Paper V*, *Paper VI*, and *Paper VII*. In Paper V and VI, we show how workload models are created following a systematic approach where we start by identifying key performance scenarios. The workload models are successively built following a few basic guidelines. In Paper VII, we describe an automatic approach of inferring a workload from historical data. The Paper presents the algorithm and explains each step of the algorithm in detail. This work was validated by conducting two experiments. In the first experiment we showed that the tool can generated a workload model from a *Apache* web server access log. In the second experiment we set out to test if the generated workload model is in fact correct. We manually built a workload model, generated load out of it in order to obtain a data log. We then used that log to generate a new workload model which we then compared to the original one. The result of the experiment was that the two workload models were close to identical.

1.5.2 Increasing the Quality of Models Used for Testing

Our second contribution is on increasing the quality models used for testing. Since model are used for test generation, the generated tests will be on the same level of quality as the models. By increasing the quality of the model, we can also increase the quality of the tests and, hence, avoid unnecessary mistakes. This contribution is divided into two parts; one approach for increasing the quality of models used for functional testing and another approach for increasing the quality of workload models used for performance testing. This contributions answers research question **RQ 2**: How can the quality of the test models be increased?

In our first approach, we propose a solution for *increasing the quality of UML models by using OCL validation*. We provide a set of custom rules and pre-defined set of validation rules for increasing the quality of the models. The rules check that the UML conform to certain principles and that important data is not omitted. The corresponding research is presented in *Paper III* and *Paper IV*. The papers describe the validation procedure and discuss the validation rules for checking different aspects of the models before proceeding to the test generation phase. The papers also describe the OCL rules, how they can be invoked, and what happens when a rule is violated. The research work has been validated by executing our pre-defined OCL rules against UML models describing the behavior of a tele-communications network element.

In our second approach, we describe an approach for *increasing the quality of PTA models*. The research on increasing the quality of PTA models is presented in *Paper VI* and *Paper VII*. In Paper VI, we present the MBPeT tool and the capability of checking the models for various properties before proceeding to the load generation phase. In Paper VII, we present an automatic approach, with tool support, for generating workload models from web server log data. Rather than constructing a model based on estimates, the models generated from log data actually represent how real users interact the system. In performance testing, it is important that the load generated from workload models mimic the load generated by real users as closely as possible, otherwise it is not possible to draw any reliable conclusions from the test results [23]. Given a set of web server log data that describe historical access patterns, the tool will infer a workload model. Our results show that the algorithm is both fast and accurate when inferring a workload model. The work has been validated by conducting two experiments. In the first experiment we create workload profiles using real log data from a web site maintaining sports scores. In the second experiment, we demonstrate that the created workload profiles actually conforms to the data found in the logs by comparing the automatically created profiles with profiles built manually.

1.5.3 Requirements Modeling and Traceability Across an MBT Process

Our third contribution of this thesis is on modeling and tracing requirements across a model-based testing process. This contribution is divided into two parts; one approach for tracing requirements in a functional MBT process and another approach for tracing requirements in a performance testing process. This contributions answers research question **RQ 3**: How to model and trace requirements across a model-based testing process?

Requirements traceability is very important in software testing. It ensures that the specified system requirements are covered by tests and it provides mechanism for detecting untested requirements. In the context of model-based functional testing this is facilitated by being able to trace an individual requirement to generated tests and back to models. Our first approach is for *tracing requirements from and to UML system models*. This research is explain in more detail in *Paper I* and *Paper II*. In Paper I, we show how requirements are traced across an entire MBT process and in Paper II, we show requirements traceability can be used to prioritize test cases. The work was validated by illustrating an example where requirements were linked to system models describing the behavior of a Mobile Switching Server (MSS). More specifically, the models described different functionality of the MSC, such as, *location update*, *phone calls*, and *call handover*. The models described the previous mentioned MCS features using both 2G and 3G technologies. The example also showed how the requirements are traced to tests and later back to the models again.

In our second approach we provide mechanisms for *tracing Key Performance Indicators (KPIs) to individual user requests*. More specifically, average response time values are associated with every action in a workload model and monitored during load generation. This research is explained in more detail in *Paper V* and *Paper VI*. Besides monitoring just response time values, our proposed approach also monitors other KPI values such as throughput, CPU, memory, disk, and network utilization. The work has been validated by applying load to an auctioning web service while monitoring the above mentioned KPIs.

1.5.4 Load Generation from Workload Models

Our forth contribution involves *generation of load from workload models described using the PTA formalism*. This contributions answers research question: **RQ 4**: How models for load generation can be created and used? We present a model notation language and explain the PTA formalism. We also propose a model-based approach for generating load from user behavior models described as probabilistic models. This research is explained in

more detail in *Paper V* and *Paper VI*. The research work has been validated by generating load from models describing the behavior of user of an auctioning web site. The load was applied in real time to the web server and the response times were measured. We preformed two experiments where synthetic load of 300 concurrent user was generated and applied to an auctioning web site in real time. In the first experiment, we showed how the tool can not only monitor Key Performance Indicators (KPIs) but can also give indicators for potential bottlenecks. The second experiment served as validation for the first experiment were we showed that the proposed bottleneck by MBPeT tool in fact was the right one.

1.5.5 Tool Support for MBT

Our last contribution is related to the *implementation of the required tools to support the previously mentioned approaches*. This contribution answers research question: **RQ 5**: How can tool-chain connectivity be improved? In this contribution, we present the *MATERA tool-set*, the MBPeT tool, and the Log2Model tool, respectively. Tool support has been discussed in *Papers I-VII*. MATERA is a tool-set that helps the tester during the model creating phase and guides him/her to follow our proposed approach. In [24], we provide additional information regarding transformational support for MATERA. The papers discusses in detail how models expressed in UML are transformed for test generation. The research work on MATERA been validated by using the MATERA tool-set on an industry case study taken from the telecommunications domain. In the case study we modeled the the location update, phone call, and handover functionality of a Mobile Switching Server (MSS). We demonstrate how the models are validated and transformed for validation. Further, we also show how the MATERA tool-set support test reporting and back-tracing of requirements using data from real test execution logs.

MBPeT is a cloud based performance testing tool. We show how we use PTAs to model user profiles to generate synthetic workload. The load is generated in real-time and applied to the system under test, while measuring several Key Performance Indicators (KPIs). The papers focus on the MBPeT tool support by showing the tools distributed architecture and dynamical allocation of new generation nodes. We also show how the tool can be adapted for a cloud environment. The research work has been validated by conducting a case study and running the tool in a private cloud while generating synthetic load for an auctioning web site in two different experiments.

Log2Model is a tool for inferring workload models, expressed as PTAs, from a set of web server log files. The tool outputs a workload model that corresponds to the access patterns discovered in the log data. The tool

has showed to be both fast and accurate when generating workload models. The work has been validated by conducting several experiments in where we compare generated models to models built manually.

Below is table showing the original publications and which research questions they address.

Publication	Research Question
Paper I: Tracing Requirements In A Model-Based Testing Approach	RQ3,RQ5
Paper II: Including Model-Based Statistical Testing in the MATERA Approach	RQ3,RQ5
Paper III: MATERA - An Integrated Framework for Model-based Testing	RQ1,RQ2,RQ5
Paper IV: Applying Model-Based Testing in the Telecommunications Domain	RQ1,RQ2,RQ3,RQ5
Paper V: Model-based Testing of Web Services Using Probabilistic Timed Automata	RQ1,RQ2,RQ4,RQ5
Paper VI: Performance Testing in the Cloud using MBPeT	RQ1,RQ3,RQ4,RQ5
Paper VII: An Automated Approach for Creating Workload Models From Server Log Data	RQ1,RQ2,RQ5

Table 1.1: Original publications and the research questions they address.

Our work is split into two parts, functional and performance testing. However, the foundation and corner stones in both parts remain the same, that is, we use models for both functional as well as for performance testing. For functional testing, we develop a set of UML models starting for from the requirements of the system. The models are created following as set of guidelines and are in a later stage transformed as input for test generation tools. System requirements are also modeled and tracked throughout the whole process. For performance testing, a set of PTA models are also derived, either manually or from log data. The PTA models are probabilistic in nature and describe how the user interact with a system. These models are used as input for load generation and performance testing of the system under test.

1.6 Overview of Original Publications

In the following section we will briefly describe each original publication in turn.

1.6.1 Paper I: Tracing Requirements in a Model-Based Testing Approach

This paper we present a model-based testing approach where we show how requirements modeled in the UML language are linked/traced to other UML system models and elements that implement the requirements. We present a hierarchical decomposition of requirements where each level in the hierarchical structure relate to different abstraction levels in the system model. We also show how requirements linked to model elements are trace to executable tests and how they are traced back to the models again after test execution to visualize which requirements that were not properly tested.

1.6.2 Paper II: Including Model-Based Statistical Testing in the MATERA Approach

In this paper we build upon the previous one by using statistical data from test execution combined with UML modeling to prioritize test cases and, thus, giving us a slight control over how tests are generated. This way we make sure that the most important requirements and, consequently, the most important test get tested first. To achieve this we give each requirement an initial priority. The priority describe the importance of the requirements. By calculating a combined total, every test case can be ordered based on their total priority value. Once test have been executed, we analyze the test log and automatically update the priority value of requirements that were attached to failed test cases. This means that test cases covering these requirements will have a higher total priority value in successive test rounds and, thus, be ordered differently.

1.6.3 Paper III: MATERA - An Integrated Framework for Model-based Testing

The paper describes our proposed model-based testing process together with the MATERA tool-set. The paper puts emphasis on the tool and presents the graphical user interface. In the paper, we explain how models are created, how requirements a modeled and traced to model elements, how the models are validated and transformed for test generation. We also show how the tool supports test reporting by collecting information from test execution logs and comparing the test purposes encoded in test scripts against the results of the test execution. Finally, we present a solution for tracing

requirements back to models. We show how requirements are tracked back to the specifications from which the corresponding test cases have been generated. This facilitates identification of possible faults in the specifications.

1.6.4 Paper IV: Applying Model-Based Testing in the Telecommunications Domain

In this book chapter we describe the entire model-based testing process applied to a tele-communications case study and present a complete picture of the entire tool chain. We give a full description of all models together with their subsequent validation and transformation phases. We explain the test generation platform and show how tests are generated based on specific test generation criteria and later translated into executable tests using industrial tools. We also describe the test execution platform, how tests are executed, and how test execution logs are produced. The book chapter is the result of collaboration between many industrial partners that have come together to build an entire tool-chain. The work presented in the chapter clearly shows the applicability and usefulness of the proposed model-based testing approach.

1.6.5 Paper V: Model-based Performance Testing of Web Services Using Probabilistic Timed Automata.

This paper presents a systematic approach for creating models that can be used for synthetic work load generation. The main idea of the proposed approach is to make use of probabilistic models for describing user behavior and, thus, generate realistic load patterns that mimic real user behavior. The proposed models conform to the probabilistic timed automata (PTA) formalism and we show how the models are created and what they mean. We also show how load is generated from the models and how different performance indicators are monitored during a test run.

1.6.6 Paper VI: Performance Testing in the Cloud using MBPeT.

This article we give a complete picture of our proposed model-based performance testing process. We also present the *MBPeT* load generator that generates work load from load models representing user behavior described as PTAs. We describe MBPeT's scalable architecture that has been built to fit in a cloud environment and highlight the main features of the tool. We show how workload models are created and present various stages of load generation. Finally, we present the results of a few experiments on load testing an auctioning web service and we discuss test reporting.

1.6.7 Paper VII: An Automated Approach for Creating Workload Models From Server Log Data.

This paper describes an alternative approach for automatically generating workload models from web server log data. We describe the different steps of the algorithm for generating a workload model and also presents tool support. Finally, we show the usefulness of the approach by applying on a few experiments.

1.7 Research Setting

The research work presented in this thesis has been developed in conjunction with several large scale research projects. Most of the research presented on model-based functional testing was done within the context of the Deployment of Model-Based Technologies to Industrial Testing (D-Mint) project [25]. D-Mint was a international project consisting of 24 partner from 5 different European countries. The aim of the project was to develop methodology, tools and training material and to adapt model-based testing into an industrial reality to cut the cost of developing high quality, complex software.

The rest of the research in this thesis has been done in collaboration with 3 large scale software projects. First, the Cloud Software Finland project [9] was a joint research project with over 30 IT companies in Finland and many other universities. The aim was to improve the competitive position of Finnish software intensive industry in global market through cloud software solutions. Second, in the PAM project [26], the objective was to investigate the applicability of model-based testing principles in the context of continuous integration processes. Finally, the Need 4 Speed (N4S) project is a consortium including a mix of over 30 partner ranging from large industrial organization to research institutes and universities. The goal of the project is to develop a foundation for the Finnish software intensive businesses in the new digital economy.

I was also part of the doctoral programme on Software and Software Engineering (SOSE) [27], where I had a funded position for two years. The programme focuses on research problems related to the software engineering process, software design and architecture, software implementation, and software testing. The research covers the study and development of methods and tools used for the various phases of software development and emphasizes practical applicability of the results to be evaluated in industrial contexts.

The findings presented in this thesis is largely the result of much collaboration with industrial partners and research institutes in the above mentioned projects.

1.8 Structure of the Thesis

The thesis is written as a collection of peer-reviewed articles and consists of two parts. Part I provides a overview of the research and explains the concepts and background information in general terms, while Part II presents the original publications. Part I is structured as follows: Chapter 2 presents an overview of our approach of modeling for test generation. The chapter also presents tool support for the proposed approach and shows the results of our MBT process applied to a industrial case study. In chapter 3 we introduce our approach for modeling for performance testing and present the research done on workload model generation from various sources. We also present tool support for the proposed approach and show the MBT process applied to a case study. Finally, in chapter 4 we present the conclusions and discuss future work.

Chapter 2

Modeling for Functional Testing

This chapter we introduce a few basic concepts, and presents the research work related area of creating models for functional testing. We also discuss the related work and present our approach for modeling for functional test generation

2.1 Background

In this section we will present the foundation of our approach and discuss a few relevant concepts and tools. The concepts and the tools are presented because they are used in our approach and are essential for understanding how our approach works.

2.1.1 The Unified Modeling Language

The Unified Modeling Language (UML) [28] is a general purpose modeling language. A general purpose modeling language usually means that almost any kind of system can be modeled using that language. The opposite to a general purpose modeling language would be a domain-specific modeling language. Only specific types of system can be modeled using domain-specific modeling languages and the concepts are closely tied to the operating domain of the system. UML was designed to give a standard way to visualize the design of object-oriented software systems. Today, UML is a de facto standard modeling notation language that provides mechanisms for modeling systems in an abstract manner and representing the systems form different types of views [28]. In many cases, UML models are used as part of the system specification document and due to its large user base it facilitates communication between systems designers and developers.

To represent a system from different viewpoints, UML deploys a number of different diagrams. Each diagram shows the system from a particular point of view. A *class diagram* presents the systems from a static point of view. It shows all the classes/modules, their attributes and functions, and how they are interconnected. The *state diagram* presents the dynamic view of the system. It uses state machines to view the behavior of each class/module and what input/output that is required to change from one state to another. *Sequence diagrams* are used to represent the communication between two or more classes/objects while *use case diagrams* show how different *actors* use different functionality (described as use cases) of the system. The actors often represent users of the systems but can also represent other software components. *Object diagram* are closely related to other class diagrams in the sense that objected diagrams are a particular instantiation of a class diagram. For example, imagine a system where students enroll to different courses. The class diagram would show two classes connected to each other; a class representing a student connected to a class representing a course. Using the class diagram as a basis, the object diagram would show what particular students that are enrolled to what particular courses.

2.1.2 The Systems Modeling Language

The Systems Modeling Language (SysML) [29] is also a general purpose modeling language and targeted towards the application of systems engineering. It is a UML profile that represents a subset of UML with extensions. The SysML profile also introduces a few new diagrams. The *requirements diagram* visualizes system requirements which can be linked/traced to other model elements that implements them. Requirements can also be derived by other requirements. The *activity diagram* describes the behavior of a system by showing the data flow and the control flow between different system activates. The *block diagram* is similar to the UML class diagram and show the system structure in terms of components (blocks) together with their attributes and relationships to other components. To model the internal structure of a block/component, SysML deploys an *internal block diagram describes*. This diagram describes the internal structure of a block in terms of its *parts*, *ports*, and *connectors*. The *parametric diagram* defines parameters and mathematical expressions for different block and describes how parameter affect each other when their values are changed. The parametric diagram also describes how different block depend on other system properties that they are bound to.

2.1.3 The Object Constraint Language

The *Object Constraint Language* (OCL) is declarative language for specifying rules that apply to UML models and elements [30]. Today is has become part of the UML standard. The language provides designers with constraints that can be used to query objects or put limitations on them. It is very useful in model validation for assuring the quality of the UML models via validation rules written in OCL.

2.1.4 NoMagic MagicDraw tool

MagicDraw [31] is a visual UML modeling tool developed to facilitate design and analysis of object oriented systems [31]. Besides modeling, the tool also offers support for code generation and has a open API that allows for plugin development. This way developers can extend the capabilities of the tool to suite their own needs. It also offers support for validation of UML via the OCL [30] validation engine. The tool is targeted towards professional system developers and software architects and is one of the leading UML tools on the market.

2.1.5 Qtronic and the QML Modeling Language

Qtronic[1] [32] is a tool for automatic model driven test case design which generates test cases from a model representing the SUT. The models are specified using a formalism called *Qtronic Modeling Language* (QML). QML is essentially a mix of UML state machines and a subset of *Java*. QML is an object oriented language and is used for describing different aspects of the SUT like data, data structure, behavior, input/output ports, etc.

Test cases are automatically generated from models representing the behavior of a SUT. Hence, the tool uses a black-box testing mechanism for generating tests. The behavior of the SUT is usually described using UML state machines together with QML as the action language. In QML, data are defined as `records`. Records are user-defined data types that can contain variables, methods, and operators. Hence, the records are the definitions of specifics data types used in the system and they are used to communicate with the environment. QML uses `ports` to define interfaces. An interface defines ports that can be used to send and receive data/records. The ports and the records that are exchanged constitute the interface of the SUT to its environment.

Qtronic provides support for requirement coverage during test generation. Requirements are associated to state models, more precisely to actions on transitions that implement a requirement. Basically, requirements are

[1] Now known as Conformiq Designer

tags that are used to trace if a specific action in the state model has been covered by the generated test cases.

2.1.6 The Nethawk EAST tool

NetHawk EAST [33] is a simulation platform for simulating and testing of telecommunication systems. The EAST platform consists of all the required tools needed for a complete behavior simulation of a different telecommunication technologies like LTE, 3G, WiMax, etc. It supports a wide range of testing capabilities from functional testing to performance testing. The platform uses of the Testing and Test Control Notation 3 (TTCN-3) [34] language for specifying and executing complex test cases.

The platform offers a wide range of editors for specifying test cases for the lowest level up to whole test suits. In most test case scenarios, only a few relevant message parameters need to be tested. However, in order to test these parameters, a lot of other detailed information must be provided. This added complexity could easily become a problem for tester. To address this problem, the EAST platform uses *reference libraries*. A reference libraries contains a collection of messages at different protocol levels we want to send and receive during a test session and contains pre-defined values for most message parameters. This way tester can focus on the relevant parameters without have to be an expert in every protocol.

2.2 Contributions

In this section we are presenting the contributions related to functional model-based testing. The presented contributions are described in more detail compared to the overview given in Chapter 1.

2.2.1 MATERA: A Systematic Modeling Process

MATERA is our proposed UML-based modeling approach for *systematic development of system models for automatic test generation*. A systematic approach *ensure that models are created in a step-by-step manner* and, therefore, is more structured than an ad-hoc approach. As stated in the introduction, models allow one to *focus on the relevant part of a problem and hide away implementation details*. The basis for choosing UML as the modeling language for our approach was that is a modeling language wide a very wide user base and known to many. It is also used extensively in the industry. Another reason for choosing UML is that we wanted to show that system design models are not only suitable for design but can also be used for deriving test cases.

Figure 2.1 shows an overview of a typical MBT process and the parts of the process that MATERA addresses. As stated in the introduction, test generation and execution has for the most part already been solved. Hence, the MATERA approach focuses on modeling and how to create system models in a systematic way.

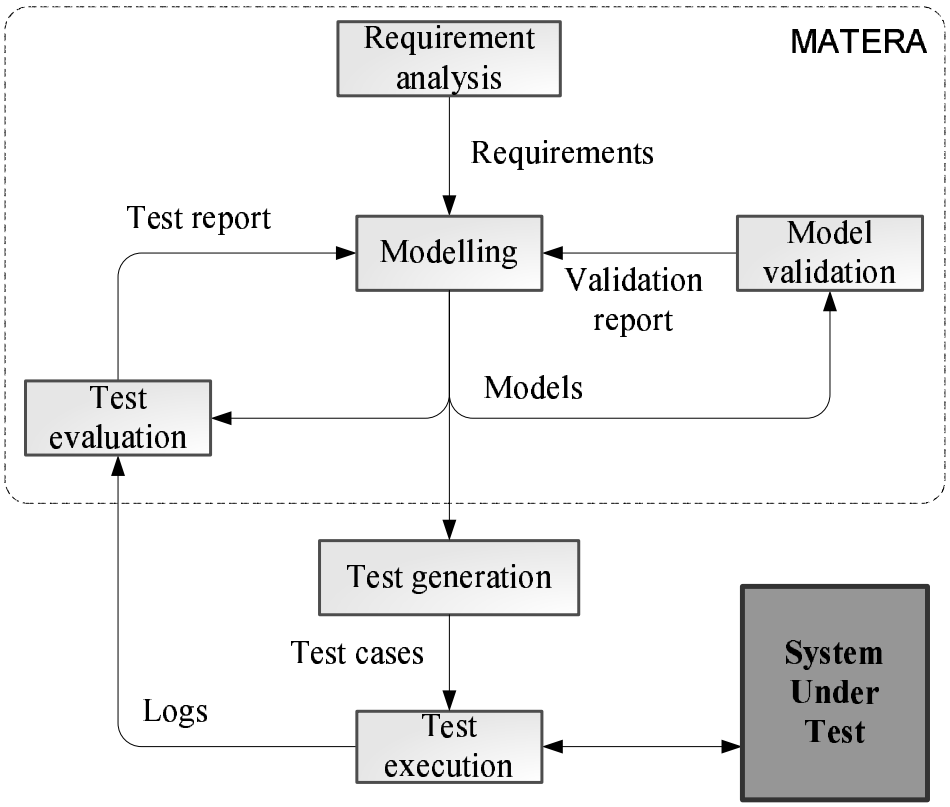


Figure 2.1: Overview of the MATERA and the MBT process.

In the MATERA approach, system models are created in a systematic fashion. Figure 2.2 depicts the MATERA process. The process is divided into five stages. First, we start by reviewing standards and specification documents and we analyze the stakeholder requirements related to the product. The time required to complete this phase depends on the amount of information that has to be analyzed. In the second phase, we construct a feature diagram and a requirements diagram based on the information gathered from phase one. The feature diagram (Figure 2.3) depicts different characteristics of the system and shoes how they are decomposed into sub-features. The idea is that every feature maps to a high level system functionality requested by a customer. For every feature, a requirements diagram (Figure 2.4) is constructed.

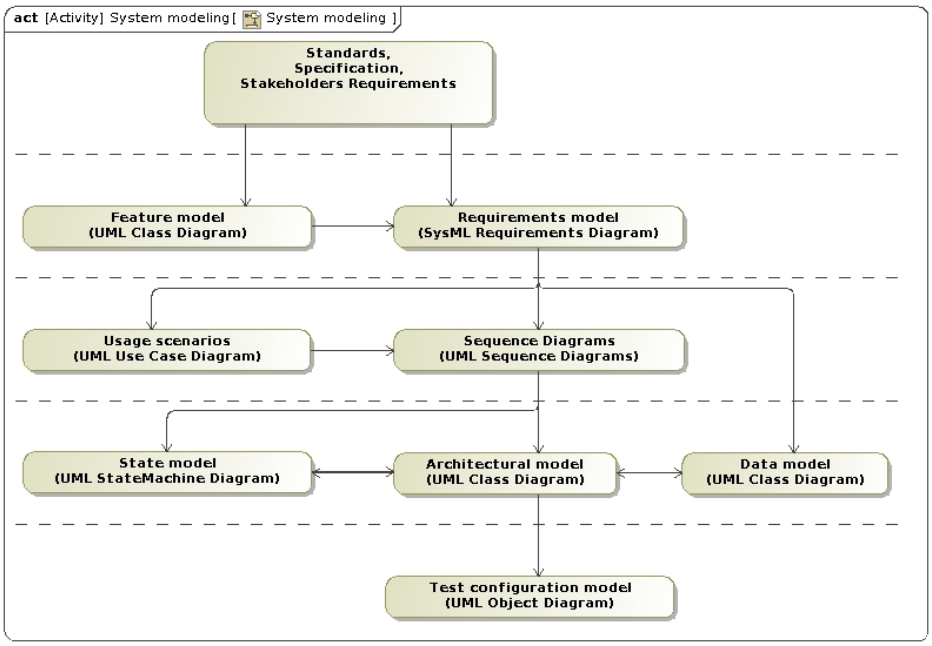


Figure 2.2: The MATERA Modeling Process.

The requirements diagram capture all the system level requirements and presents a general overview of the requirements. We start by modeling a single requirement which maps to a system feature. From here, the requirements are decomposed into sub-requirements based on the information gathered from the first phase. This process in repeated until we reach requirements that are testable or no further sub-requirements are identified. The requirements can also be given a priority value. The priority value indicates the importance of a requirements and is in later stages of the testing cycle used to order test cases based on their importance. The priority values are considered to be given from external sources (e.g., system requirements or stakeholder recommendations) and known a priori before the first iteration of the testing process. In later cycles, the priority values can be adjusted based on statistics of untested requirements from previous test cycles for targeting the testing process towards a certain part of the SUT.

In the third phase we construct use case diagrams and sequence diagrams from information contained in the requirements model. The use case diagram (Figure 2.5) specify how system functionality is provided to its environment. It shows how users access different system functionality, represented as a use case. Every use case have a detailed description of pre- and post-conditions, sub-cases, and a tabular format for specifying the necessary message communication between actors and the systems in order to com-

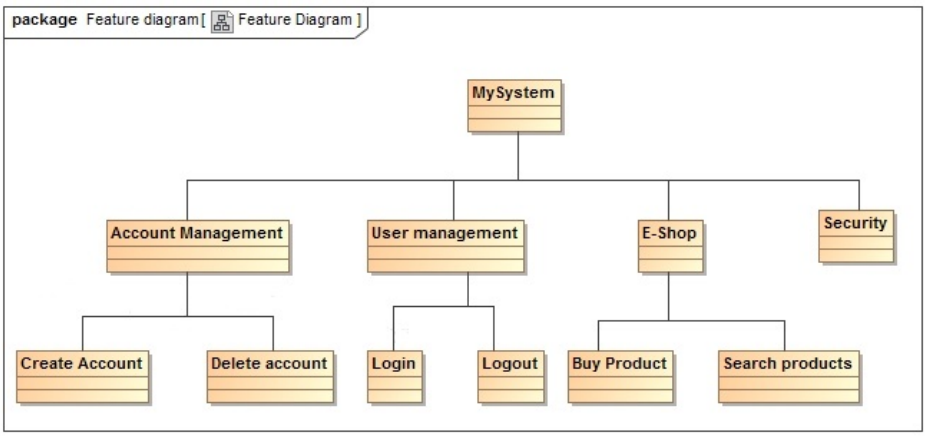


Figure 2.3: Example of a feature Diagram.

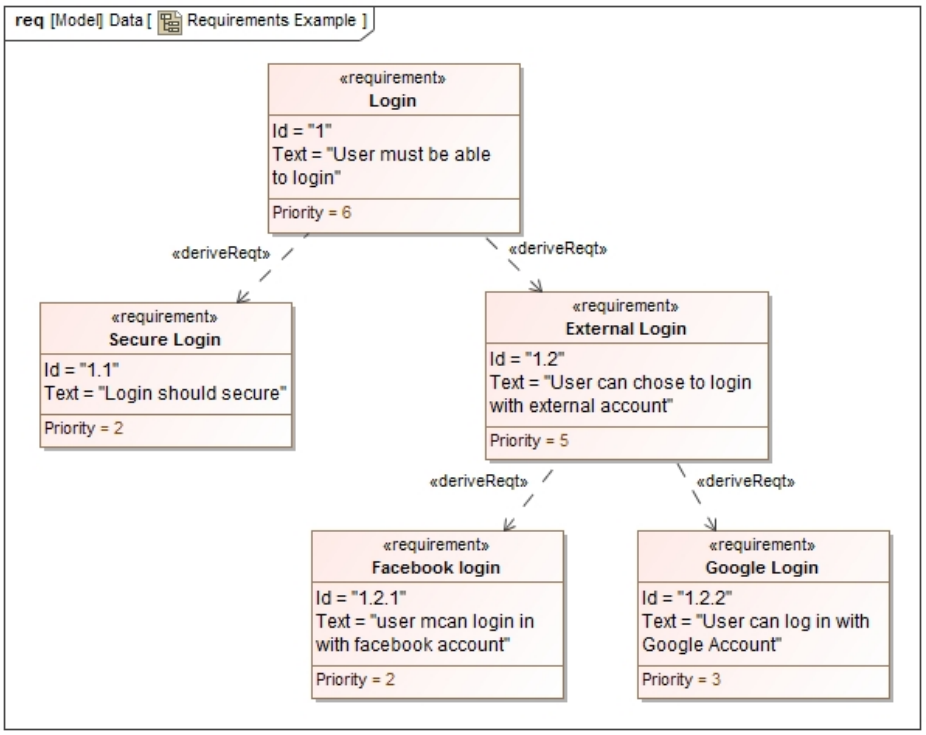


Figure 2.4: Example of a requirements diagram.

plete a use case. This process is repeated for every identified use case. Use cases also have a probability value that indicates the chance for a user requesting that functionality from the system. The probability value together

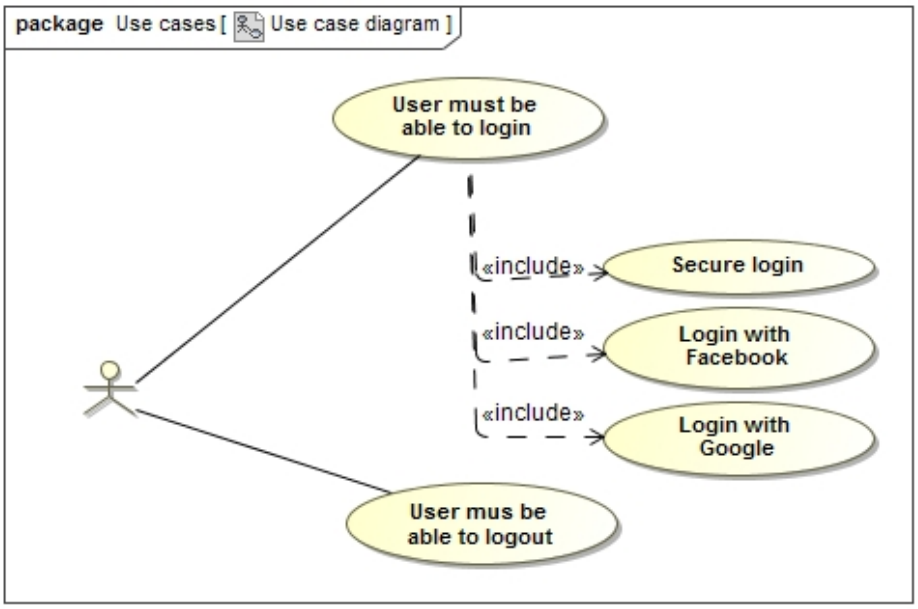


Figure 2.5: Example of a use case diagram.

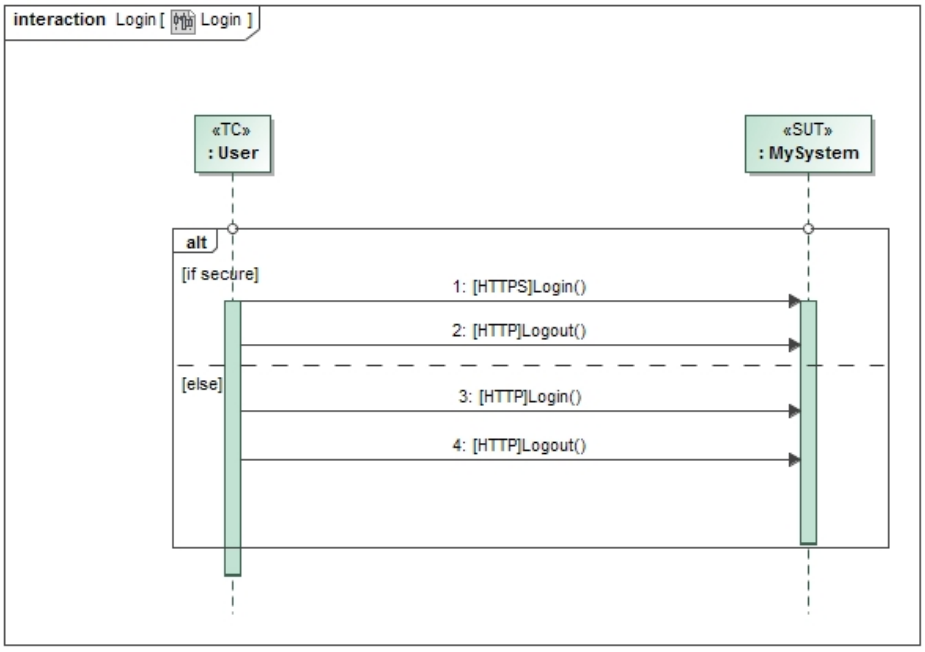


Figure 2.6: Example of a sequence diagram.

with priority values for requirements make up the back bone for ordering test cases based on their importance. For every use case, a sequence diagram (2.6) is constructed. The sequence diagrams primarily describe the interactions between system components in a sequential order. They can also show the messages that are sent between actors participating in a use case. The exact order of the message exchange is normally extracted from various protocol specification documents and standards.

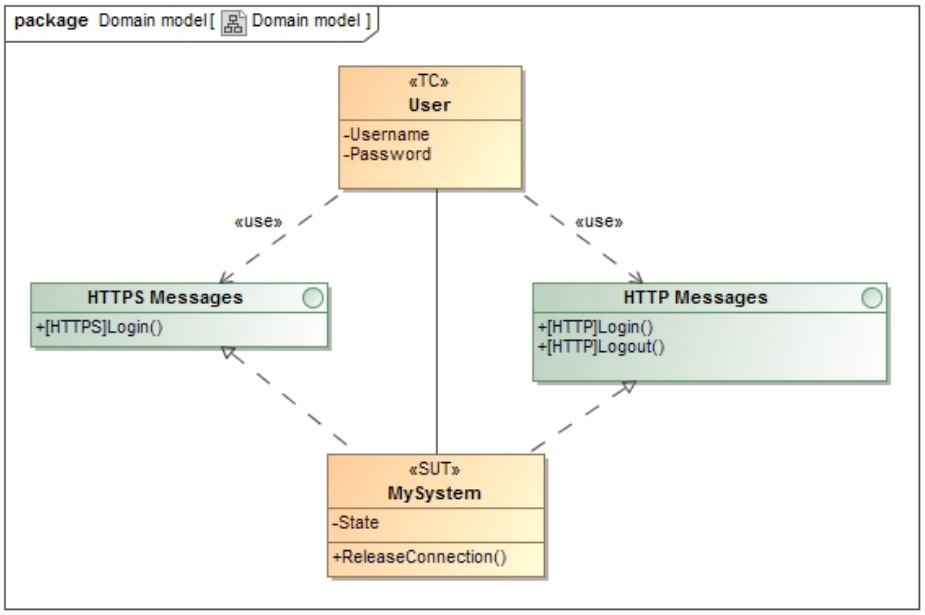


Figure 2.7: Example of a domain model.

In the fourth phase, we create a set of models that describe the architectural and behavioral parts of the system. From this we introduce three different models, a *domain model*, a *state machine model*, and a *data model*. The domain model (see figure 2.7) depicts the system and how it is connected to other components. The domain model also shows the interfaces used for communication between the components. Every interface have two channels, one for sending and one for receiving messages. The domain model is derived from sequence diagrams and is built iteratively.

Every object found in the sequence diagrams is represented as a class in the domain model. The interfaces are obtained from the messages sent and received by different object in the sequence diagrams. For every class in the domain model we construct a state model (Figure 2.8). The state model describes the behavior of an entity in terms of states and transitions. A transition can be seen as an event where the system is either sending or receiving an input and, thus, resulting in switching from one state to

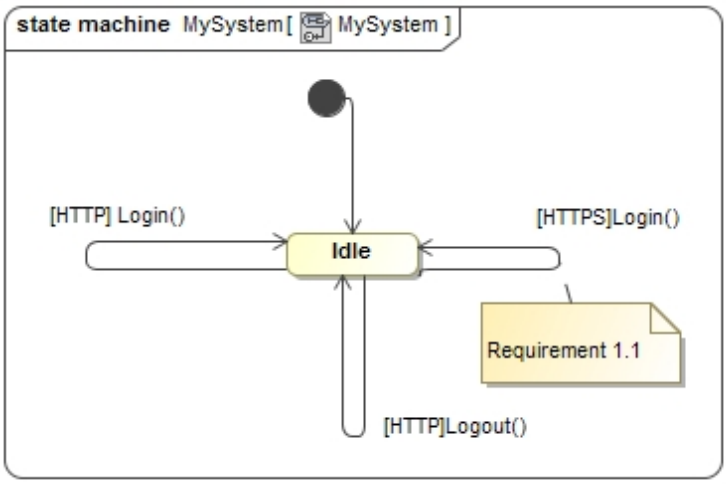


Figure 2.8: Example of a state machine model.

another. By overlapping the message comminution each sequence diagram a state model is constructed for each system component. The resulting state model may also contain composite states. A composite state is simply a state containing other states and is a way to abstract or reduce the complexity of a state model and make it more readable.

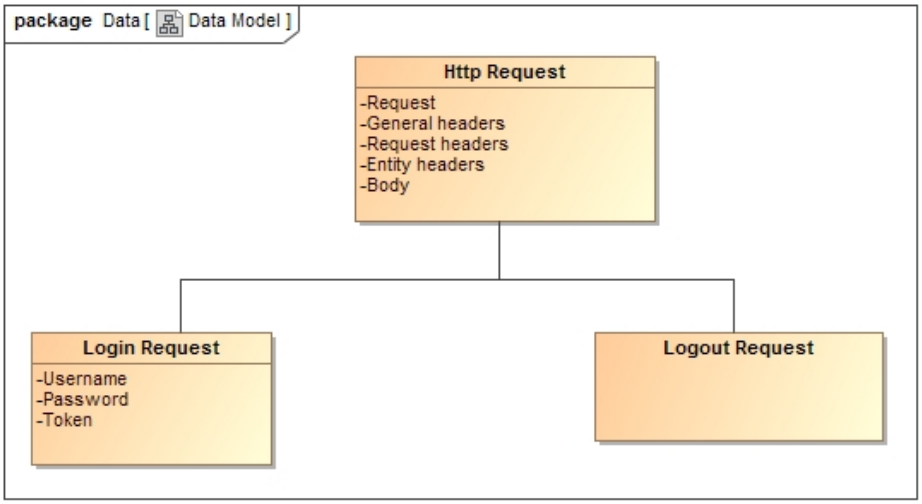


Figure 2.9: Example of a data model.

We also need a description of the data used in the system. For this purpose, we introduce a data model (see Figure 2.9). This model depicts

the messages related to a specific protocol and we use inheritance to model common parameters. Each message is represented by a class and the message parameters are modeled as class attributes. The data model is built via an iterative process in where we gather information from protocol standards and various requirements documents. Every class in the data model is linked to a message in an interface in the domain model. This is done to ensure traceability throughout the whole process.

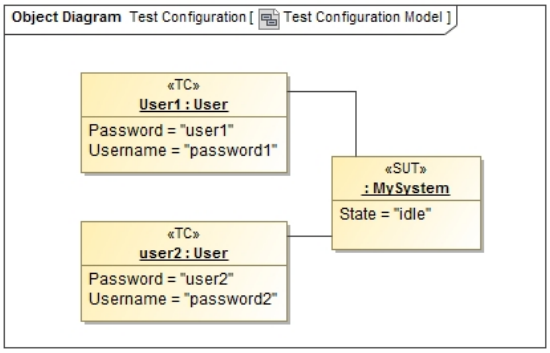


Figure 2.10: Example of a test configuration diagram.

In the last phase, we employ a test configuration model. This model is an instantiation of the domain model and shows a particular configuration at a particular time of the test environment. For example, Figure 2.10 shows a system under test (SUT) connected to two test components, TC1 and TC2. All of these object are instantiations of different components. At this time, different class attributes or test parameters get their specific values.

2.2.2 Increasing Model Quality through Model Validation

For increasing the quality of models, we present a *model validation approach*. Since tests are going to be generated from models, *increasing the quality of the models also increases the quality of the generated tests*. During the validation stage, models are check for *consistency*, *correctness*, and *completeness*. With consistency we mean that the models are consistent with respect to each other. For example, that information in one diagram does not contradict the information specified in another diagram. With correctness we mean well-formedness, i.e., that models conform to the UML standard. For example, whether the target item is a valid UML item, or whether a diagram contains valid elements within it. We notice that this check is left to the UML tool that we are using, since most modern UML tools will not allow users to break the UML meta-model. However, some tools might allow users to specify models that are not well-formed. With completeness, we mean

that the models are complete with respect to the transformation and test generation process. For example, we check that a model contains certain diagrams and that these diagrams contain the right elements with right values and stereotypes. More information regarding model validation can be found in [35]. In general, the concepts are related to checking that our UML models conforms to certain guidelines, the UML standard, and that all the necessary data is present in the models. The validation step is a prerequisite for proceeding to test generation because with low quality models only low quality tests can be generated.

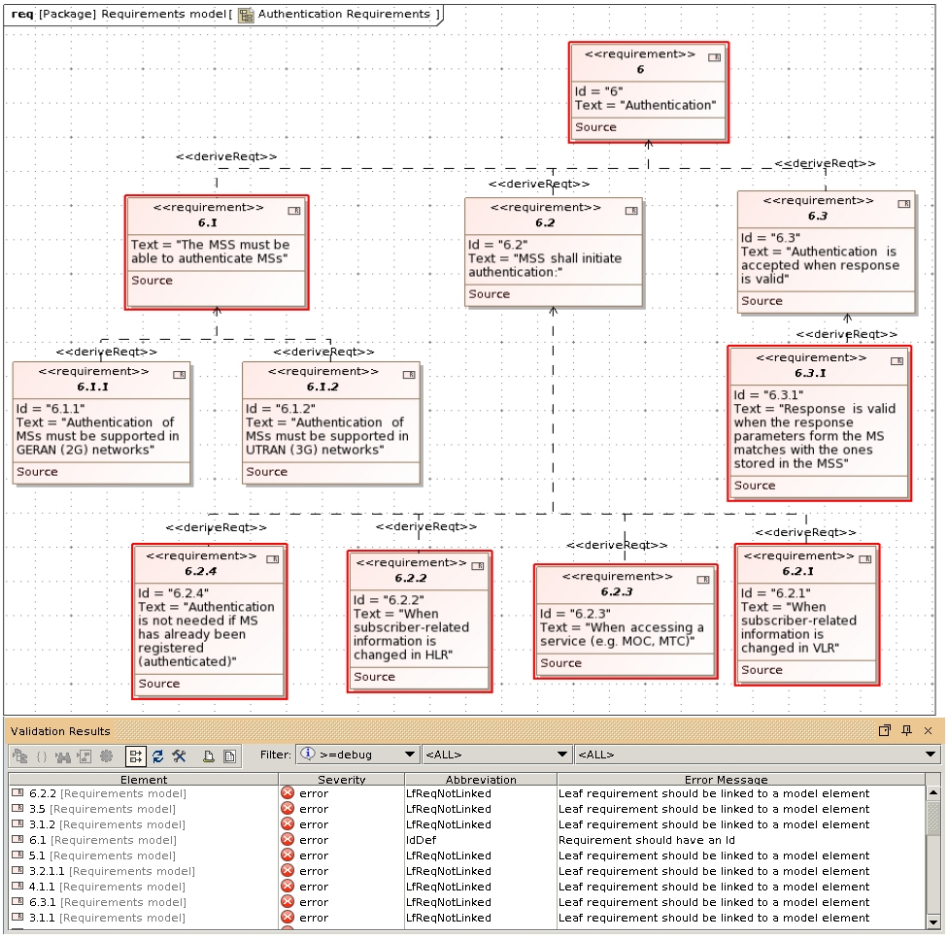


Figure 2.11: Example of a model validation.

The validation process allows us to check, for instance that all leaf requirements are traced to different model elements and that no requirement is overlooked. It also allows us to check that all the data fields are properly filled in. Figure 2.11 depicts the result of validating a requirement model

in MagicDraw, according to our specified rules. The *Validation Results* box shows all the OCL constraints that has been violated during the validation process. In this example, MagicDraw detected that requirement 6.1 has no *id*. All the requirements highlighted in red are not traced to any model elements. This means that those requirements will not be propagated further to any tests. Before proceeding to the next phase of the process, all violations need to be corrected and models re-validated.

2.2.3 Requirements Traceability Across the MATERA Process

Requirements traceability is essential to the MATERA process. The activity is preformed throughout the whole testing phase in order to *assure that all requirements have been tested*. To accomplish this requirements need to be traceable both to and from tests. Gotel and Finkelstein define requirements traceability as "the ability to follow the life of a requirement, in both forwards and backwards direction, i.e., from its origins, through its development and specification, to its subsequent deployment and use" [36].

One of the main purposes of tracing requirements to models is for *analyzing which parts of the specification "implement" different requirements*. This will allow later on propagating these requirements from models to tests. Another reason for tracing requirements is that if a requirement changes, it is *essential to know how this change is reflected in the models*. To be able to verify what requirements that have been tested, we trace requirements from models to test runs and back to models as illustrated in Figure2.12.

As shown previously, we create requirements diagrams that describe how requirements are decomposed into a tree-like structure. In our research we employ a specific relationship between requirements and other model elements. The relationships between requirements and models elements are specified on several levels. Non-leaf requirements are linked to models, e.g. state machine models. An exceptional situation is in the case of top-level requirements which are linked to use cases in the use case diagram. The leaf requirements diagram are linked to other model elements to which they apply, e.g. transitions in a state machine or classes in a class diagram. This is done to ensure traceability of requirements to test cases. When the state machine is later executed, if a transitions is successfully executed we consider the requirement tested.

These links are useful for evaluating whether all the modeled requirements have been reflected in the models. Further, by tracing requirements to model elements we can trace requirements, which were left uncovered during testing, back to system models again. This facilitates the process of identifying which of the requirements that have been left untested.

Once the models have been created and all the requirements liked to

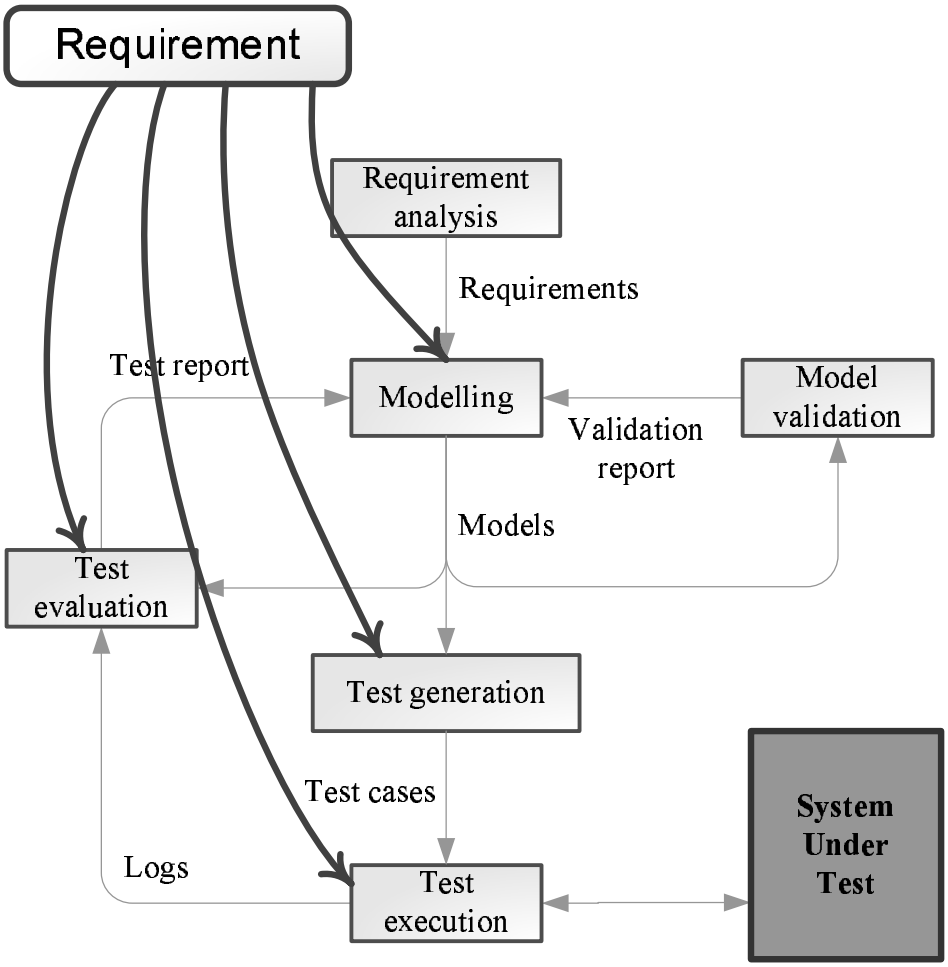


Figure 2.12: Tracing requirements throughout the MBT process.

model elements, the models undergo a transformation phase. During this stage the UML models are transformed into a output format suitable for test generation. This is because UML models, as such, can not be used for test generation.

2.3 Validation

In this section we describe how each contribution has been validated. We present tool support and empirical evaluation of the presented research applied in a case study.

2.3.1 Tool Support

This chapter discusses the related work and presents the research work we have done in the area of model-based testing from a tool support perspective. More precisely, we will present MATERA, a tool-set that integrates modeling in the Unified Modeling Language (UML) with requirement traceability across a model-based testing process. This chapter focuses on the framework and tool that have been developed to support our MBT process. The process is supported by a tool chain, depicted in Figure 2.17. The MATERA tool-set is built as a plugin for the MagicDraw [31] modeling tool.

Modeling. As mentioned earlier, a set of models are created in a systematic manner starting from the system requirements. The models represent the SUT from different perspectives. For creating and editing UML models we use NoMagic's Magicdraw [31]. One of the main reason for choosing MagicDraw as our modeling tool is that it offers great support for extending its capabilities by various plugins and profiles. Moreover, MagicDraw also offers support for the SysML profile. We wanted to use the benefits the SysML's requirements diagram to model requirements. The requirements diagram facilitates tracing of requirements to UML model elements. Another reason is that MagicDraw offers support for model validation. MagicDraw is delivered with an OCL interpreter that can be used to validate models for suspicious and faulty constructs. By validation the system models, errors can be detected and eliminated before moving on to the test generation phase. In this way, errors are not propagated onto later stages in the MBT process. Also, developing our own model editor was outside the scope of our research, hence, we chose Magicdraw because it is a mature tool and well known in the industry.

Model Validation. In order to gain efficiency of using a MBT process and reducing the costs by discovering faults at an early stage, the models are validated by checking that they are consistent and that all the information required by our process is included. Therefore, the MATERA tool set comes equipped with a set of modeling guidelines and validation rules for ensuring compatibility with our MBT process. In our MBT process, validation is prerequisite before transforming the models.

The MATERA framework utilizes the validation engine of MagicDraw for model validation. The engine uses OCL [30], a formal language for specifying rules that apply to UML models and elements, to validate the models. These rules typically specify invariant conditions that hold true for the system being modeled. Rules written in OCL can be checked against UML models and it can be proved that nothing in the model is violating them. UML is accompanied by several predefined suites of validation rules. Apart from the rules provided by MagicDraw, user-defined validation rules are also allowed in the MATERA framework. The author of [35] have defined

a complementary set of validation rules to ensure compatibility with our modeling process a smooth transition to the subsequent phases. In order to facilitate reuse, validation rules are stored in different validation suites depending on the intended purpose of the rule. The validation suites can be invoked at any time during the model creation process. Upon invocation, each rule will be run against the UML element type for which it has been defined. If an UML element is violating any rule, the user is notified in a Validation Results editor (Figure 2.13). By clicking a failed rule, the elements violating the rules are presented to the user.

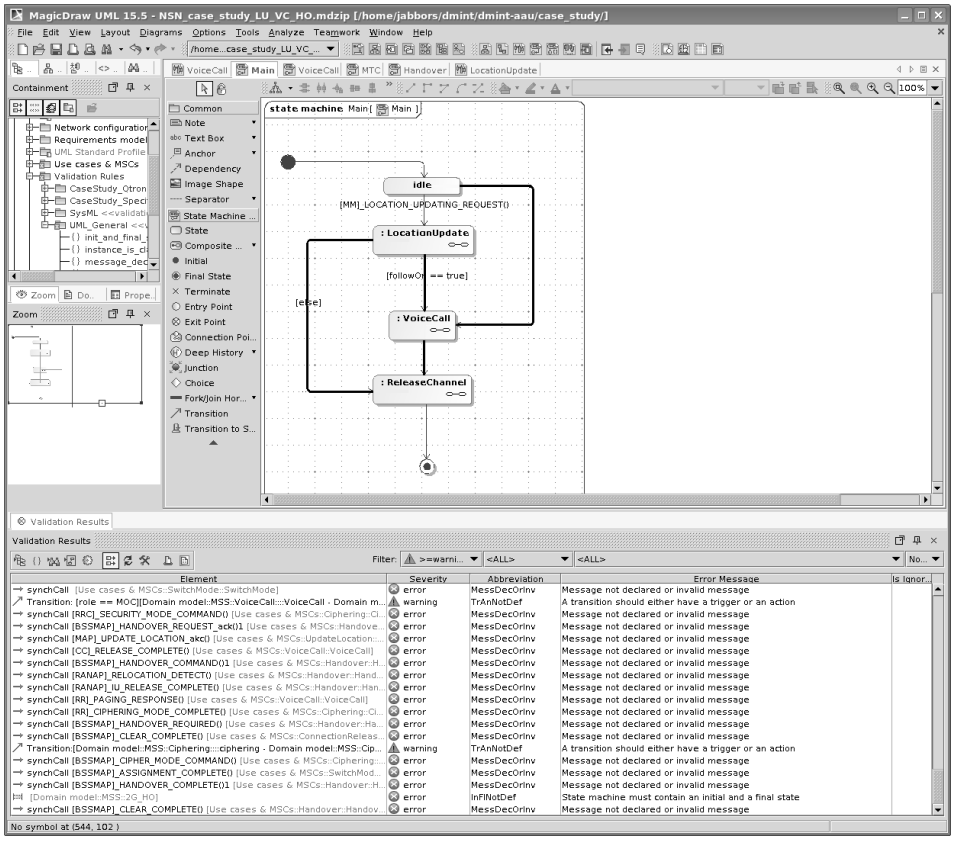


Figure 2.13: Example of MagicDraw's Validation Results editor.

Model Transformation. The created UML models representing the SUT are used for generating an input model for the Conformiqs Qtronic test generation tool [32], using the MATERA model transformation module [24],[37]. The transformation phase has two steps. First, information from UML models is gathered by a parser module and stored in internal data structures. Second, the gathered data is read, by various build modules, and later transformed in the format supported by the test generation tool,

namely QML. In our current research we have targeted only Conformiqs Qtronic test generation tool. However, the goal of transformation approach is to be generic and to be able to expand our approach to target other test generation tools when needed. During the transformation phase, requirements are also propagated from UML models to QML. In QML, requirements are treated as textual tags attached to different parts of the model. The requirements tags can later be used as testing goals during the test generations process. The total size of the transformation module is roughly 2000 lines of code.

Test Generation. In our MBT process, test are generated by Conformiqs Qtronic test generation tool [32]. Qtronic automatically derive tests from system models that represent the desired behavior of a system. Tests are generated using black-box testing techniques and so the tests evaluate the SUT according to on its external behavior, not by inspecting its internal workings. Tests are generated by identifying paths in the system model that together cover selected testing goals. A test case covers a certain testing goal if execution of the test against the model itself would cause the goal to be exercised. The testing goals range from covering requirements to transitions and states to boundary value analysis. The tool utilizes several advanced techniques for simulating the system model and generating a minimal test suite that covers the desired testing goals. By selecting different testing goals, the user can affect how Qtronic generates test cases.

Since the QML model is an abstract representation of the SUT, the generated test will be on the same abstraction level. Therefore, in order for the generated tests to be executable, they need to be brought to the same level of detail understood by the external interfaces of the SUT. In Qtronic, this is achieved using a scripting back-end or an adapter. Once attached, the back-end will render out the abstract test cases to a format understood by the SUT. The only drawback is that, in most cases the back-end have to be implemented manually.

After the test cases have been generated, the test generation tool can determine the generation order of test cases based on the annotated probability and priority values. For each generated test case, a weighted probability is calculated based on an algorithm implemented by the test generation tool. The weighted probability is calculated from both the use case probability and the requirement priority and determines the sequence in which test cases are ordered (see Figure 2.14). Test cases are finally rendered into executable test scripts using an adapter for concertizing test cases into executable scripts.

Test Execution. In order to target as many system domains as possible, we chose to have to execution phase completely de-coupled from the MATERA process. There are plenty of different application domains, test execution platforms, and languages. Instead of restricting the MATERA process to a particular execution platform and language, we decided to al-

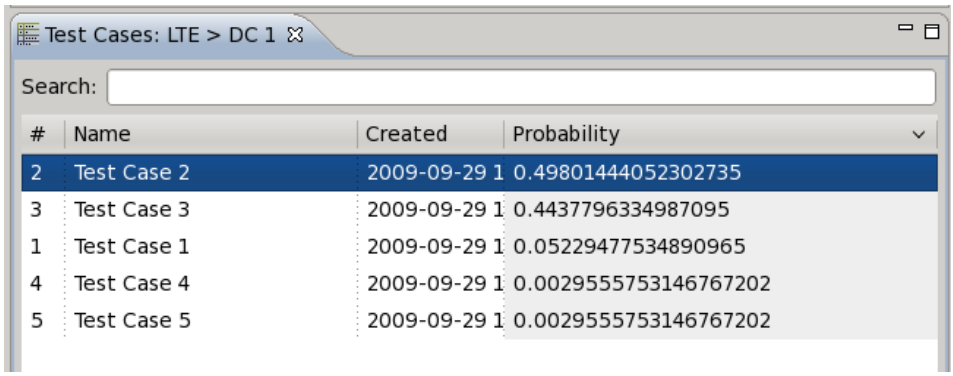


Figure 2.14: Test case sequence ordered by weighted probability in Qtronic.

low the testers to freely choose the test execution platform and language. However, a few restrictions are still enforced. The chosen test execution platform and language must support test execution logs and the annotation of requirement tags in the chosen language. This is because be use test execution logs to check which requirements that have been covered during testing.

Requirements Traceability. As suggested earlier, requirements traceability is one of the key features of MATERA and we also keep track of the requirements during the entire MBT process. In addition, tracing the requirements helps to understand the impact that new or modified requirements have on different artifacts. Altogether, the process supports shorter feedback loops that, in turn, serve as a basis for modern product development conventions such as agile development practices.

More specifically, tracing of requirements allows for requirements to be propagated to test specifications. The goal in higher level testing practices is to verify that requirements have been covered by tests. Needless to say, requirements are the keystone in any successful project implementation, and hence, they must be traceable both to models and tests. In our process, requirements are not only traced from model to tests, but also back from test to models. Tracing requirements to tests can even help in identifying missing tests, that is, where critical requirements do not trace to any test. Finally, if a test fails, one can trace the requirement back to the models from where it originated, in order to identify the error. This facilitates the process of identifying which parts of the system model cause a test to fail. For example, figure 2.15 shows how a requirement associated with a failed test is traced back to a transition in a state machine in UML. The figure shows how the element linked with the requirement is highlighted in order to locate the problem area faster.

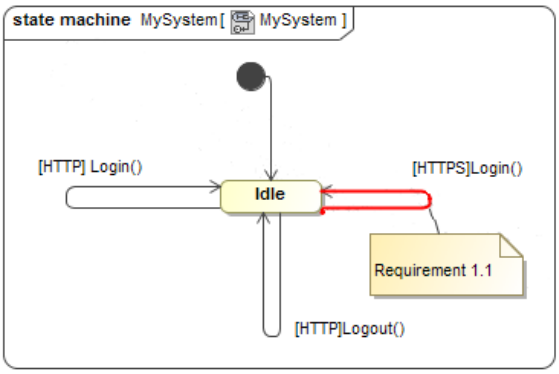


Figure 2.15: Back-tracing of requirements to a state machine model.

2.3.2 Empirical Validation on a Tele-communication Case Study

To validate our research work we decided to apply our approach on a telecommunications case study. The target system was a *Mobile Switching Server* (MSS), which acted as the system under test (SUT). In survey on model-based testing approaches, Neto et. al. [15] point out that hardly anyone is integrating MBT approaches with a software development process and that most MBT approaches lack empirical evaluation for an industrial environment. Hence, the reason for selecting the MSS as a SUT was that we wanted to evaluate our approach on a fairly complex system that is used in an industrial environment and also for investigating to what degree UML system models can be used for test generation. Particularly, we wanted to investigate the following aspects:

- How do we create UML models for test generation with as much reuse as possible?
- How are requirements traced from model to tests?
- How can requirements be traces from test to models?

An MSS is a telecommunication networks core element and it is responsible for controlling the rest of the network elements [38]. The MSS is connected to its surrounding elements via several different interfaces. Figure 2.16 shows an example of the topology of a telecommunications network.

In the figure, the central element is the MSS. It is connected to several other sub-systems e.g., Base Station Sub-system (BSS) and Radio Network Sub-system (RNS). The BSS is a 2G telecommunications sub-system and consists of several Base Station Controllers (BTC), which in turn controls several Base Transceiver Stations (BTS) [38]. The BTS is a radio tower

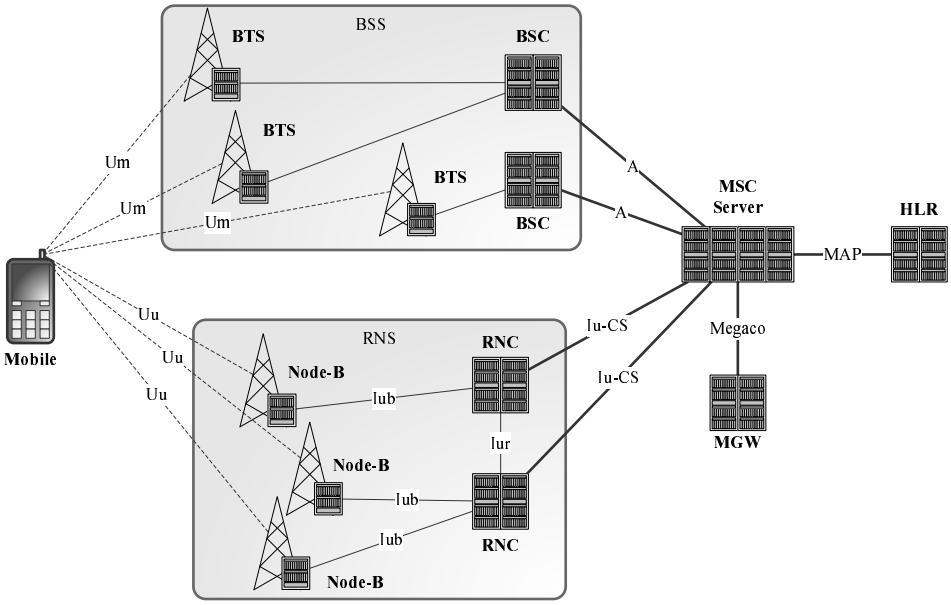


Figure 2.16: Telecommunications network architecture.

equipped with 2G (GSM) technologies for sending and receiving radio signal from various telecommunications devices. Similarly to BSS, the RNS is a 3G telecommunications sub-system and consists of several Radio Network Controllers (RNC), which in turn controls several Node-B's [39]. A Node-B is a radio tower equipped with 3G (UMTS) technologies for sending and receiving radio signal from various telecommunications devices. Most often the Node-B and the BTS are located on the same physical radio tower but are controlled by different controllers. The Media Gate Way (MGW) is a network element responsible for converting data formats from one network type to another, while the Home Location Register (HLR) is a database containing details about various mobile devices that are authorized to use the network.

The main features of the MSS that we investigated was the *location update*, *voice call*, and *handover* procedures. The location updating procedure enables devices to inform the network when they move from one location area to the next. A location area is a small geographical region covered by one radio tower. The voice call procedure is the method by which the network connects two mobile device together that wish to make a call. Handover is the procedure that enables mobile devices to move from one location area to another during an ongoing call, without disconnecting the call

Figure 2.17 presented the tool chain that was used in the case study. MagicDraw was the editor used for creating and validating our system models. The MATERA transformation module was used to export our system

models to QML notation. As a test generator, we used Conformiqs Qtronic. Using a test renderer, the generated test were exported to EAST scripts. Finally, we used Nethawk's EAST tool as a test execution platform.

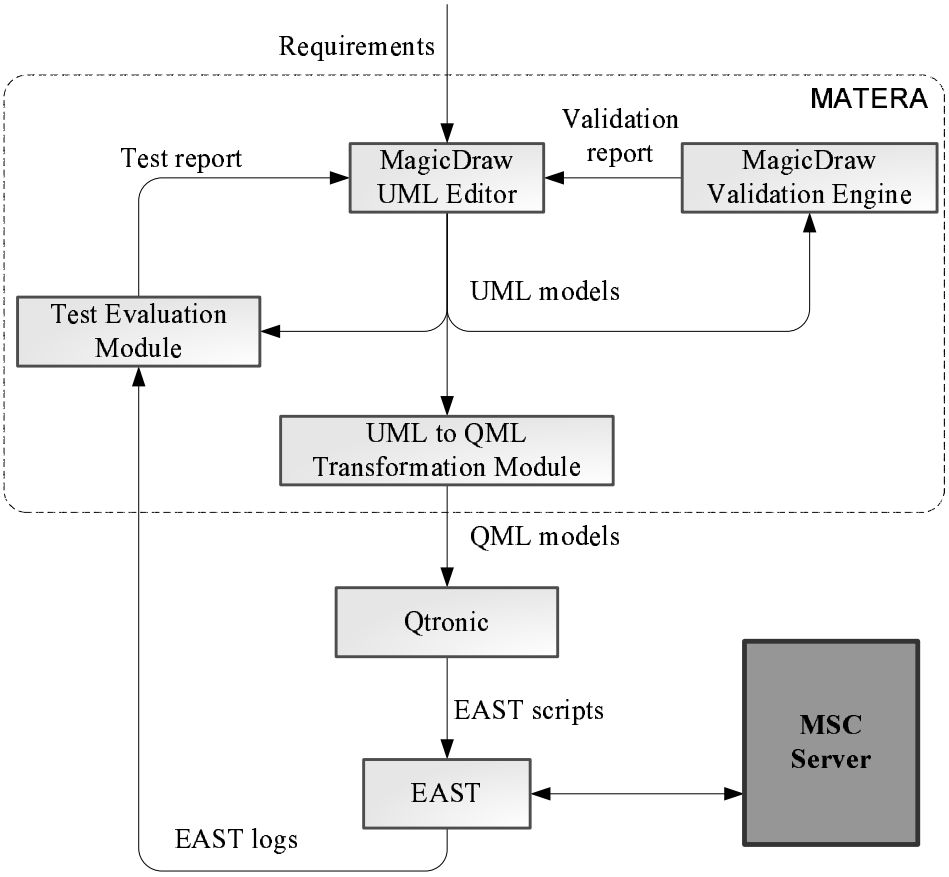


Figure 2.17: Tool chain overview.

Modeling. As mentioned above, the MSS had 3 main features, *location update*, *voice call*, and *handover*. These features should be supported both in a 2G network as well as in a 3G network. Figure 2.18 depicts the MSS feature diagram.

For each decomposed feature we created a requirements diagram. Figure 2.19 depicts one of the three requirements diagrams created.

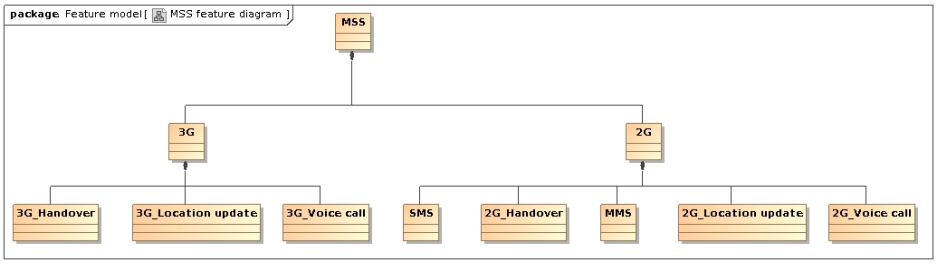


Figure 2.18: MSS feature diagram.

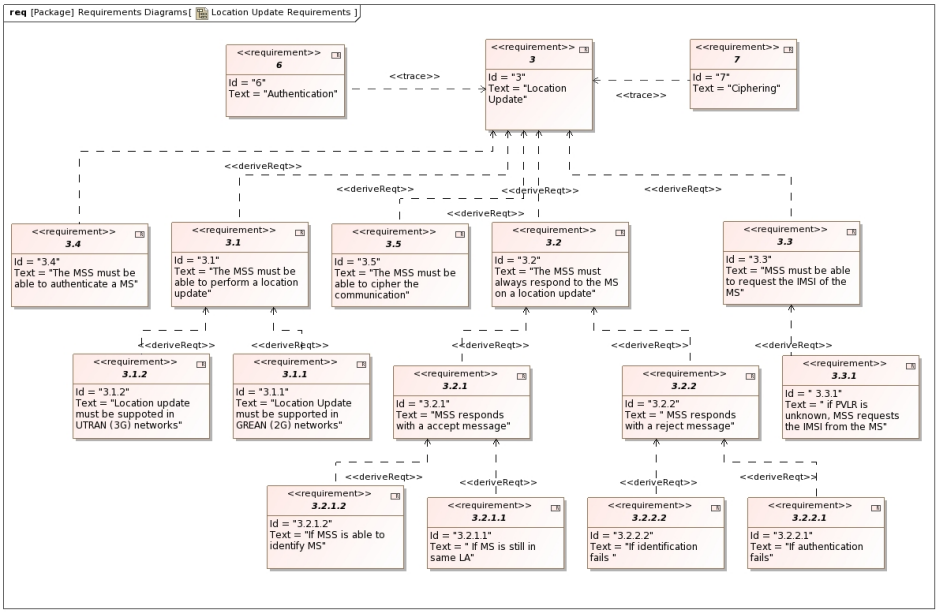


Figure 2.19: MSS Requirements diagram.

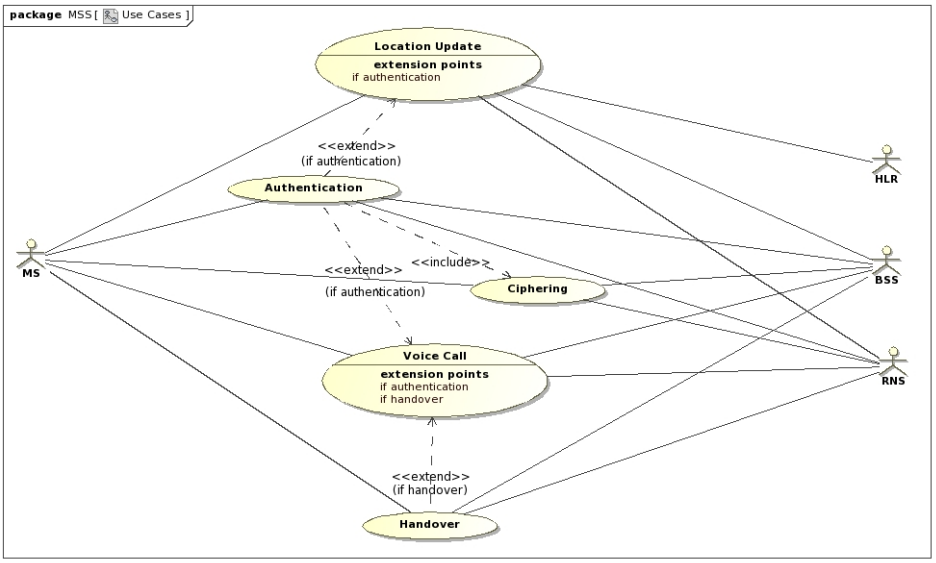


Figure 2.20: MSS use case diagram.

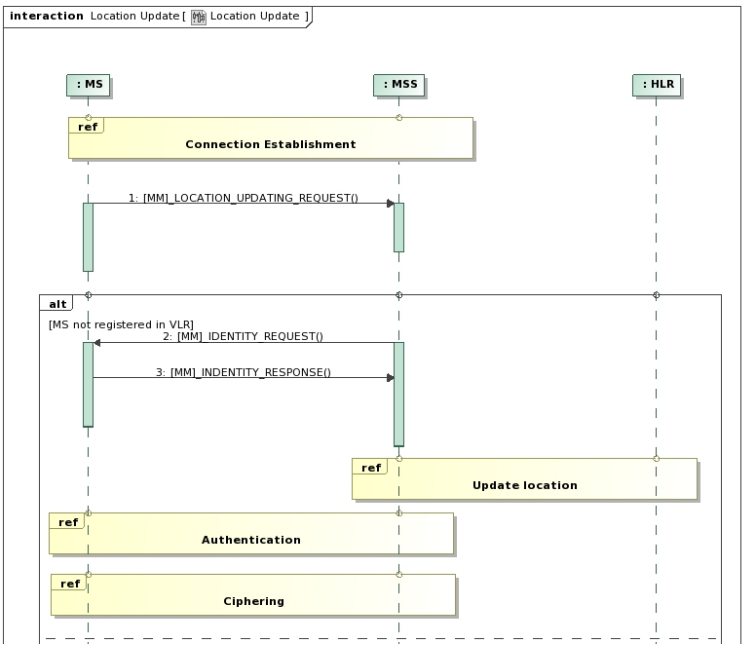


Figure 2.21: Sequence diagram for Location update procedure.

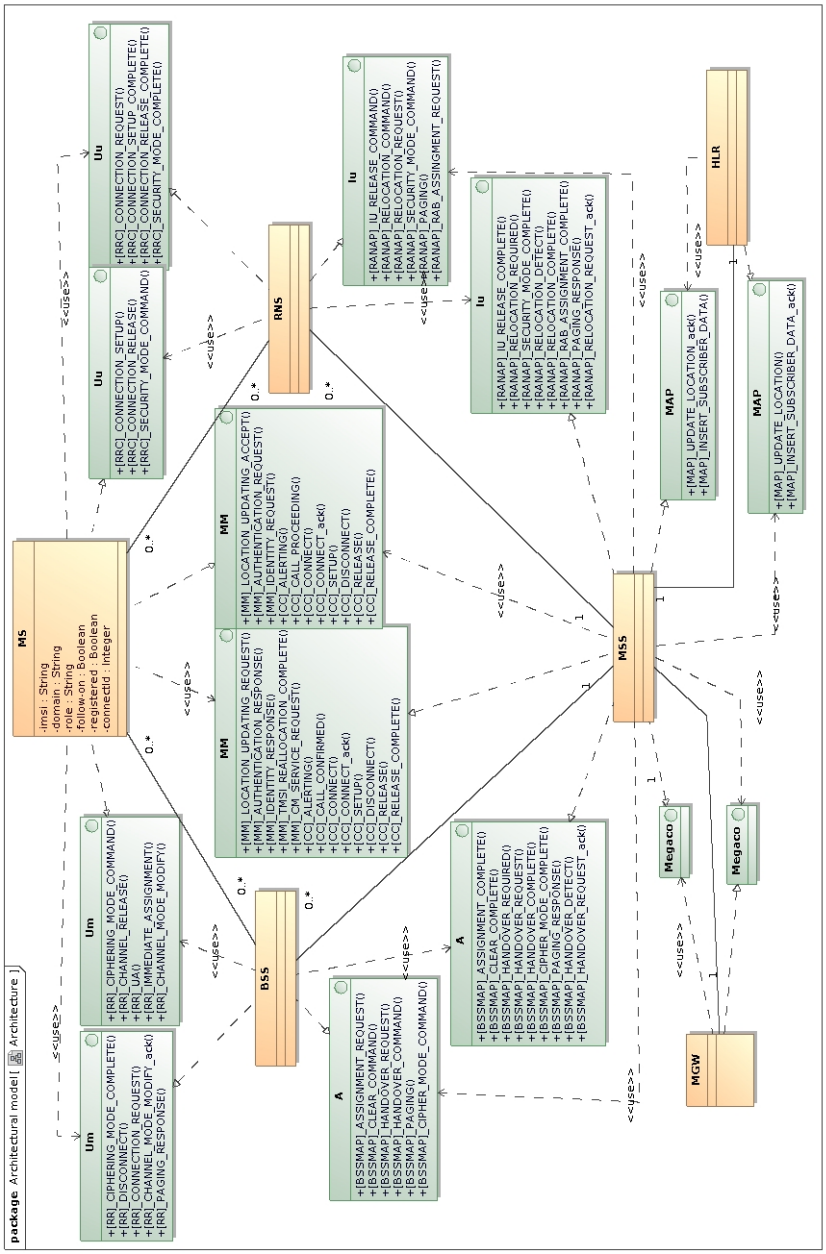


Figure 2.22: MSS domain model.

Once the requirements have been identified and modeled we created use cases and sequence diagrams for each feature. Figure 2.20 shows a use case diagram for the identified use cases, while Figure 2.21 shows a sequence diagram for the location update procedure. Similar diagrams where created for the remaining use cases.

In the next stage, we created a domain model, data models, and state machine models. The domain model shows a static view of the system under test and how it is connected to its environment. The diagram also shows what interfaces it uses for communication with the environment. Figure 2.22 shows an overview of the domain model.

For each interface in the domain, we created a data model. The data model describes the structure and parameter of the messages sent and received by the system under test. Figure 2.23 depict the data model for mobility management (MM) interface.

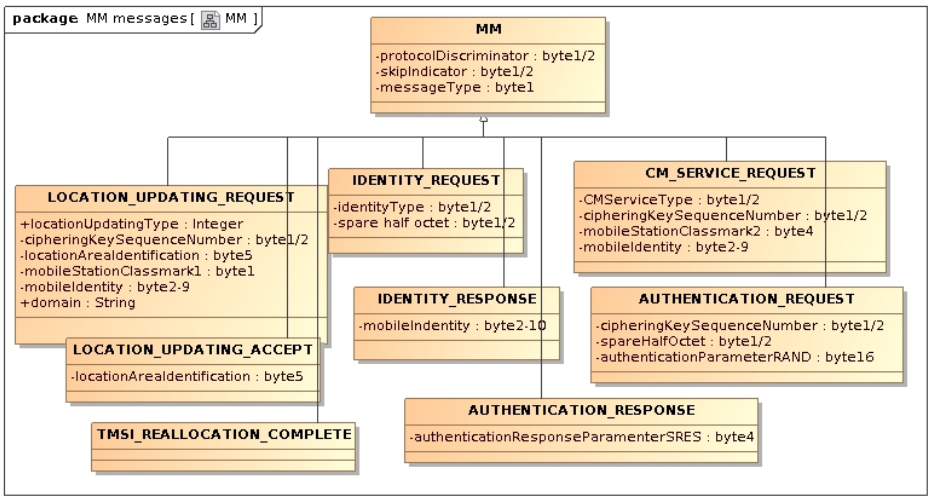


Figure 2.23: Data model for the Mobility Management (MM) interface.

By overlapping sequence diagram with each other we obtained a set of state machines. In total, we obtained 19 state machine diagrams that together describe the dynamic behavior of the MSS. Figure 2.24 show the state machine diagram for the Location update procedure. The state machine also contains sub-state machines that, for example, specify behavior for sub-routines during the location updating procedure.

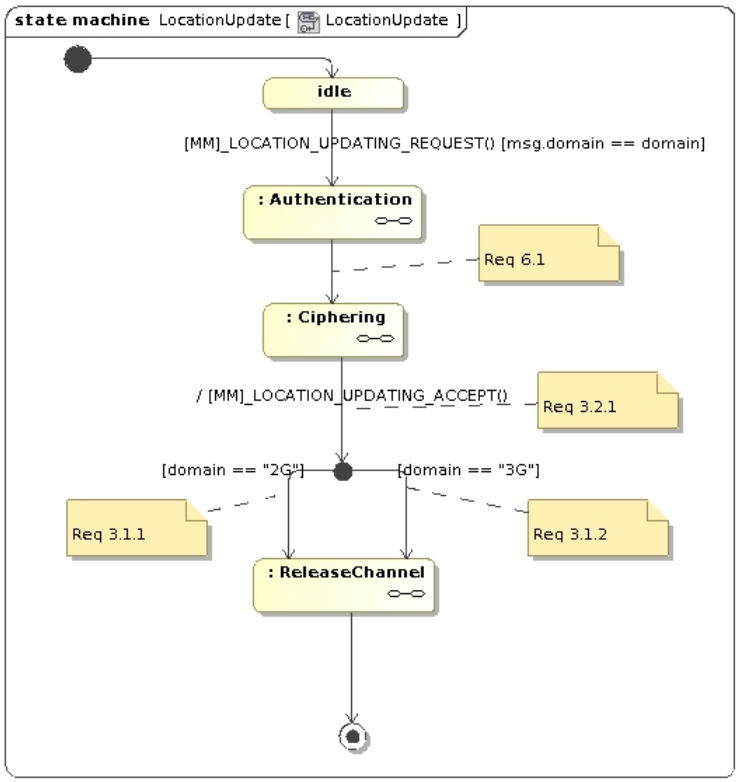


Figure 2.24: State machine diagram for Location update procedure.

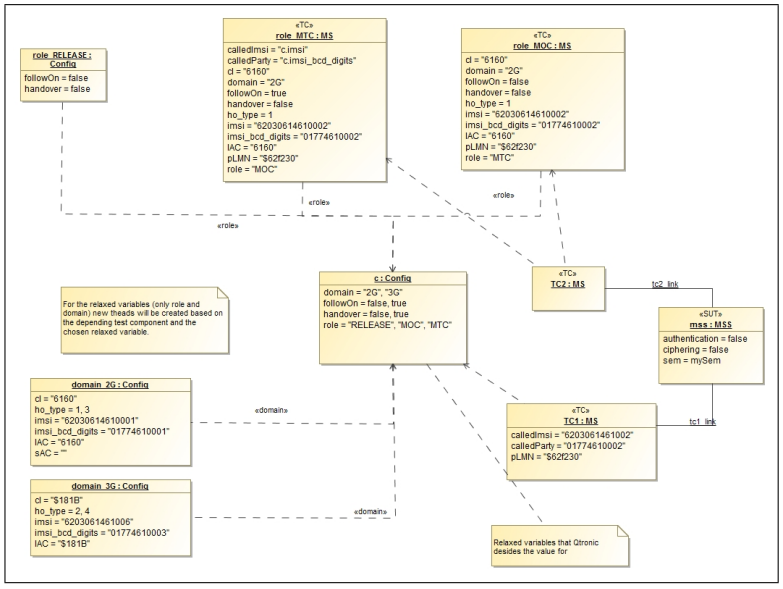


Figure 2.25: MSS test configuration diagram.

That last type of diagram type we created was the test configuration diagram. This diagram represent an instantiation of the domain diagram. For example, Figure 2.25 shows a configuration with two test components (TC) connected to one test system (MSS). The two main test components can in turn inherit parameter values from other test components.

In total we created 40 different diagrams to represent the SUT. To give a sense of scope of the modeling effort, table 2.1 below shows a list of created artifacts.

Type	Amount
Requirements	35
Use Cases	5
Sequence diagrams	9
Interfaces	10
State machine diagrams	19
Data Messages	69
States	88
Transitions	136

Table 2.1: List of modeled artefacts

Model Transformation. The entire collection of models representing the SUT from different perspectives was then transformed into QML, a modeling notation understood by the test generator tool Qtronic. Transforming the models to QML took around 1 minute. The generated QML state machine were of the equivalent size as the UML state machine, i.e., 88 state and 136 transitions. The generated JAVA part for QML comprised of approximately 700 lines of code.

Test Generation. From the resulting QML models a total of 114 test cases [40] were generated in 30 minutes. Figure 2.26 shows a picture of the Qtronic test generation tool. After the test cases have been generated, the test generation tool can determine the generation order of test cases based on the annotated probability and priority values. For each generated test case, a weighted probability is calculated based on an algorithm implemented by the test generation tool. The weighted probability is calculated from both the use case probability and the requirement priority and determines the sequence in which test cases are ordered.

The generated test cases were rendered into executable EAST [33] test scripts using a scripting backend. The scripting backend is a piece of adapter code for translating abstract test cases into executable test scripts. The scripting backend had to be implemented manually and consisted of roughly 2000 lines of code. The executable tests were of the size 200-400 lines of code, depending on the amount of test step in each test. This clearly shows the

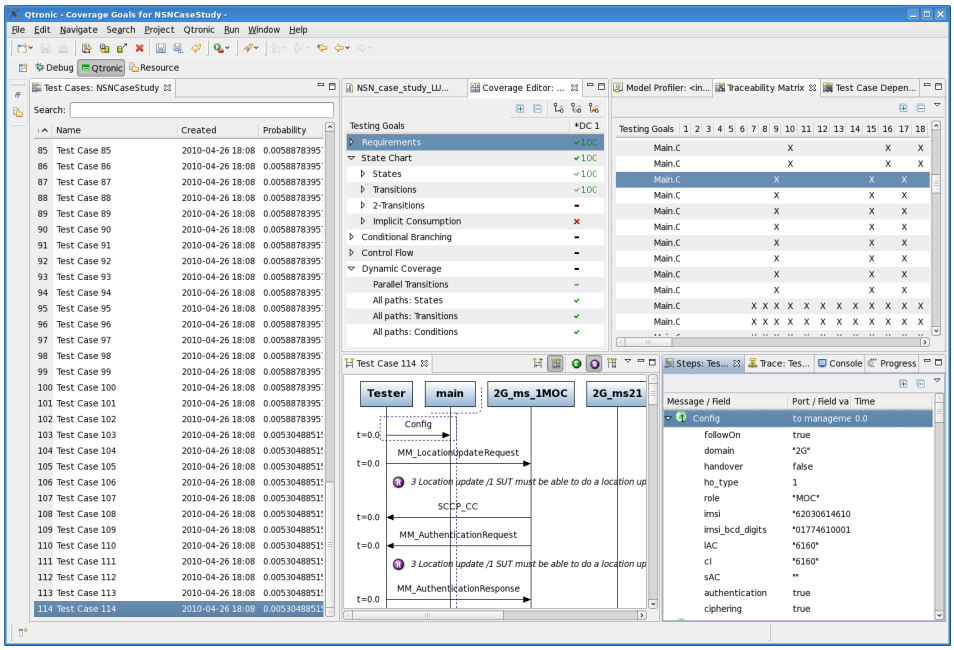


Figure 2.26: Screenshot of the Qtronic test generation tool.

benefit of MBT over having to manually write tests.

Test Execution. The test were executed using NetHawk's Environment for Automated System Testing (EAST) platform. While EAST is executing tests, the output of each test execution is stored in EAST test execution logs. These logs contain detailed information of every test step, executed method, covered requirements, etc. We use this information to keep track of which test that failed during execution and the requirements that were linked to such tests.

Requirements Re-Propagation. The information gathered from the test execution logs is used to trace requirements back to UML models and to update the priority value of those requirements were left uncovered during test execution. In other words, requirements that were linked to test cases that failed during testing. The update procedure is done by incrementally increasing the priority of requirements associated with the failed test cases, such that they will counterbalance the effect that the probabilities of the use cases have on the ordering of tests. As the process is iterated several times, the priority (importance) of requirements will change according to how much they have been tested. This means that, priority values for requirements that need to be tested more thoroughly in a subsequent test iteration are incremented with a predefined coefficient and automatically updated.

In this case study, the executed test cases reveled no errors. This is mainly because, in this case study, the generated test cases was executed on

production Mobile Switch Server (MSS) and we were not expected to find any errors. With this in mind, it is problematic to show the how requirements are trace back to models when all test were successfully executed. However, after purposely manipulating test execution logs to make them look like tests had failed, we could gather information from failed test cases. Figure 2.27 depict tracing a requirements attached to a failed test case back to UML models.

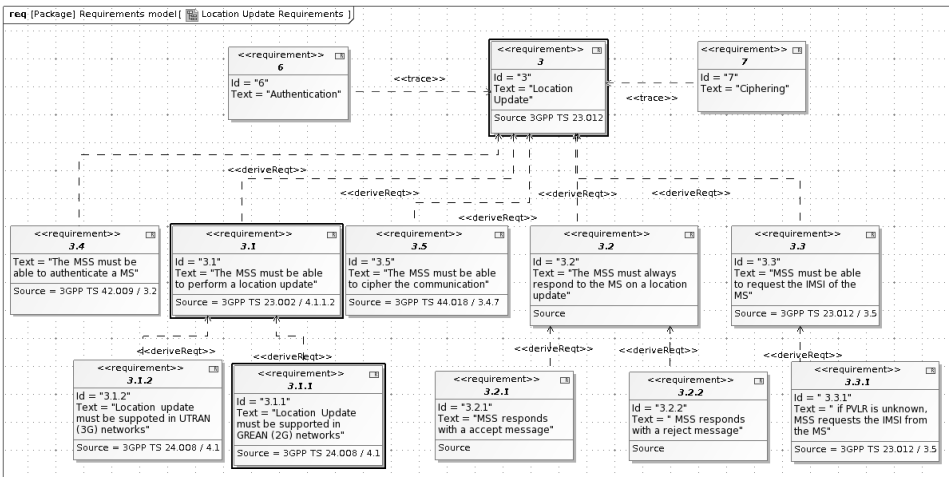


Figure 2.27: Example of tracing uncovered requirements back to models.

2.4 Related Work

As mentioned in the introduction, only about 5 percent of the existing MBT approaches use UML as the notation language. However tiny this field may be, there still exist plenty of research in this area. Most of the presented MBT approaches below offer some kind of tool support.

Binder points out that, little research has be produced in the area of testing for object-oriented system, which UML is very good at modeling [41]. Basanieri and Bertolino [42] proposes to use message sequences between objects from UML sequence diagrams and then combine it with category-partition testing [43].

Briand et. al., describe the TOTEM system test methodology [44] which uses sequence diagrams, use cases, and state diagrams to describe test models and in [45], they present and method for investigating test coverage criteria based on state charts using simulation techniques. In fact, others have also proposed to generate test sequences from state charts. Offutt and Abdurazik, proposes test criteria to generate test cases based on state charts [46], while they in [47] use traditional data-flow coverage criteria in UML

collaboration diagrams for static checking but do not present tool support test case generation. Chow is of the early proposers of using state machines to test software [48]. However novel, these approaches are all validated using text book examples with state diagrams no larger than 5-10 states and lack empirical validation outside academia.

Sinha and Paradkar, propose a MBT methodology for testing web services using extended finite state machines [49]. In a "lessons learned" paper, Dalal et. at., report on practical uses of MBT on four cases studies [50]. The authors report on the importance of test data models, having an iterative approach, using abstractions, managing change, etc. The mentioned approaches all lack one or several of these concepts.

Cavarra et. al., propose an approach of transforming UML object models into a formal description, namely the IF language, from where tests are generated [51]. Santosh et. al., propose an approach for generating tests from UML sequence and activity diagrams [52]. Tretmans and Brinksma, propose an approach, with tool support, for test generation from models expressed using formal notations [53].

There exist much more work in the area of UML and MBT and not all will fit into the scope of this thesis. The presented approaches all have their basis in UML and support test generation one way or the other. However, most presented approaches do not focus of test execution and back tracing of the results to models. Hardly anyone is tracing requirements throughout their process to be able to tell what has been covered by test and no one have empirical test results of their from the industry. Our approach extends upon these limitations and thus addresses all of these above mentioned points.

2.5 Conclusions

In this chapter, we have presented our modeling approach for modeling for functional testing. We have shown the main contributions and explained how they have been validated. We have also shown how our research have been validated in the terms of the tools that have been built and the empirical validation that was done in a case study. In the case study, a total of 40 different UML diagrams was used to represent the SUT. From these diagrams a total of 114 test cases was generated. This shows that model-based testing is applicable even to very large scale and complex industrial systems. In fact, one of the analysis of results from all the consortia in the D-mint projet showed that, not only could direct test costs be reduced by 15 percent using model-based testing, but that test coverage could be improved by 10 percent [11]. The results presented in this chapter serves as a basis for answering the questions of how to *model* for test generation and can MBT be used in an *industrial setting*.

Chapter 3

Modeling for Performance Testing

In this chapter, we present our contributions related to modeling for performance testing. We also discuss the related work and present how the research has been validated through experiments and tool support.

3.1 Background

3.1.1 Performance Testing

Performance testing is the process of testing how a system performs in terms of speed, responsiveness, stability, etc., when the system is put under a particular workload. Other quality attributes of a system can also be verified using performance testing, such as, scalability, resource usage, reliability, etc. Figure 3.1 shows a typical model-based performance testing process. For instance, a model is developed by analyzing the performance requirements, the system specifications, and various other documents, such as Service Level Agreements (SLAs). Load is then generated from the models and applied to the system. Most often, system resources such as CPU, memory, hard drive, etc., are monitored and recorded during this stage. Finally, the results are gathered in a report where one can compare the usage of various system resources to the applied workload. Performance testing can also be used for diagnostic purposes, for instance, in locating bottlenecks in the system. Performance testing includes a number of different sub-genres or techniques for determining specific characteristics of a system. Below is a list of the most commonly used techniques:

- Load Testing: Used for measuring a systems responsiveness under an *expected* workload.

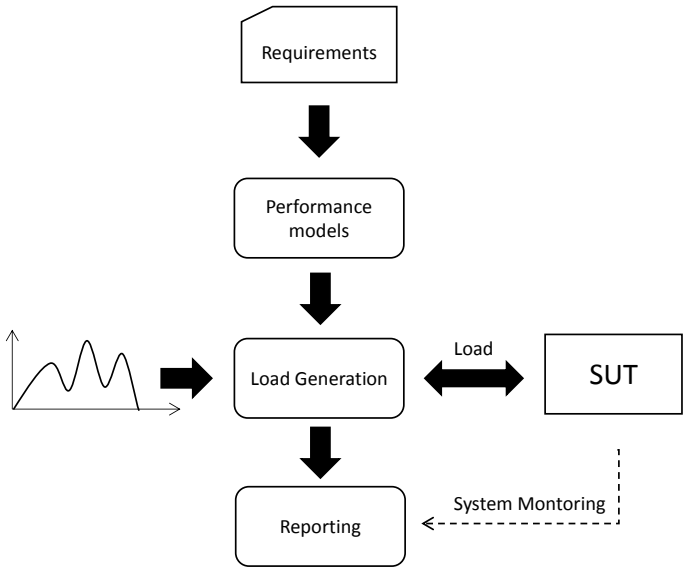


Figure 3.1: Example of a general model-based performance testing process.

- Stress Testing: Used for measuring the upper limits of a systems capacity and to determine the breaking point when a system crashes.
- Spike Testing: Used for testing a systems response and behavior to sudden large increases in workload.
- Endurance Testing: Used for evaluating if a system can sustain long periods of high load. Typically used for finding memory leaks.
- Configuration Testing: Used for testing the effects of change made to the configuration of various systems components.

3.1.2 Workload models

Traditionally, performance testing starts first with identifying key performance scenarios, based on the idea that certain scenarios are more frequent than others or certain scenarios impact more on the performance of the system than other scenarios. A performance scenario is a sequence of actions performed by an identified group of users [54]. However, this has traditionally been a manual step in the performance testing process. Typically, the identified scenarios are put together in a model or subprogram and later executed to produce load that is sent to the system.

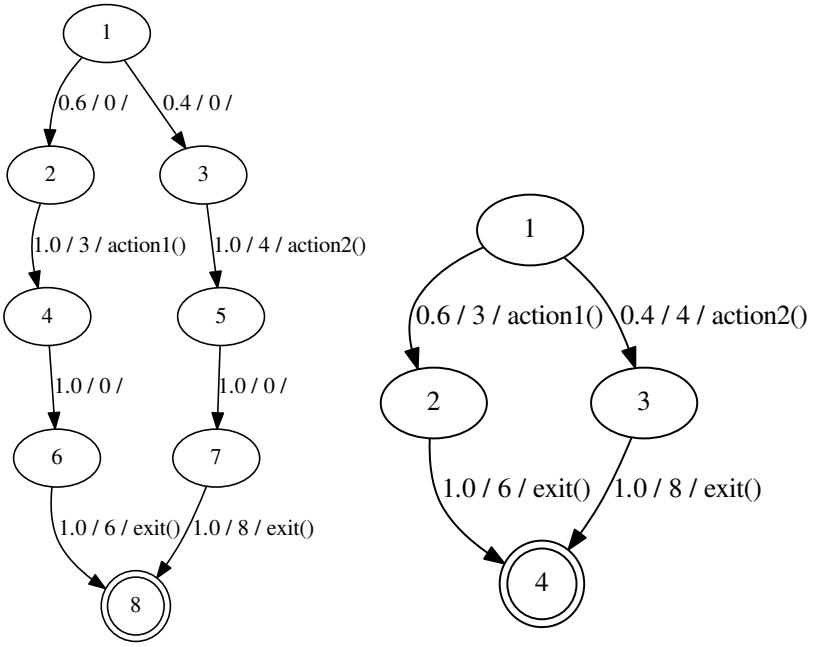
In our approach, we use *probabilistic timed automaton* (PTA) [55] to model the likelihood of user actions. The PTAs consists of a set of *locations* interconnected to each other via a set of *edges*. A PTA also includes the notion of time and probabilities (see Figure 3.2(a)). Edges are labeled with different values: *probability value*, *think time*, and *action*. The *probability value* represents the likelihood of that particular edge being taken based on a probability mass function. The *think time* describes the amount of time that a user thinks or waits between two consecutive actions. An *action* is a request or a set of requests that the user sends to the system. Taking an edge means making a probabilistic choice, waiting for the specified think time, and executing the actual action. In order to reduce complexity of the PTA, we use a compact notation where the probability value, think time, and action are modeled on the same edge (see Figure 3.2(b)).

3.2 Contributions

In this section we are presenting the contributions related to model-based performance testing. The presented contributions are described in more detail compared to the overview given in Chapter 1.

3.2.1 Distributed Load Generation from PTA Models

In our approach, we *generate load from abstract models*. Our distributed architecture allow us to benefit from the *automatic scalability of cloud environments* and our abstract models allow for *re-useability of test specifications*. We use two types of models; *user profiles* and *workload models*. Both types of models are represented using the PTA formalism. The *user profile* describe the arrival rate of different types of user to a system, while the workload model describes the probabilistic behavior of a user, i.e., how a user interacts with the system. For the moment, we are only targeting systems, i.e., web applications and web services, that use the *HTTP protocol*. This means that our workload models describe how users or other components interact with the system under test. Figure 3.3 depicts a user profile and Figure 3.2 describes a workload model. In, figure 3.3 we see a simple PTA model describing three different *user types*, their *probability*, and *waiting time*. Figure 3.2(a) shows a PTA consisting of a set of locations connected to each other via a set of edges. The values on each edge represent the *probability* of an edge being selected, the *waiting time* before firing the edge, and the *action* to execute when firing an edge. Figure 3.2(b) shows the PTA presented in figure 3.2(a) but using a more compact notation. This compact notation is used in order to reduce the size and complexity of a PTA models.



(a) Original PTA syntax (b) Compact PTA syntax

Figure 3.2: Example of a probabilistic timed automata

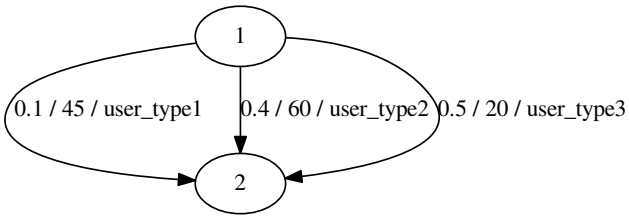


Figure 3.3: User profile describing different user types their waiting times and probability.

Whenever an edge is fired, the corresponding action is executed. Executing an action means sending a request to the system. This activity puts a small load onto the system because the system has to process the request and send back a response. Larger volumes of load can be generated by firing many edges in a short amount of time. In order to achieve this, we run many workload models in parallel. Since a workload model represents the the ab-

stract behavior of one particular type of user, by running several models in parallel, we can simulate *concurrent user* behavior. For every desired concurrent user, we executing an instance of the workload model that specifies the behavior of that particular user type. Having several different workload models, we can simulate the behavior of many different types of users. The effects of using workload models described as PTAs is that load generation is based on a deterministic choice with a probabilistic policy. This introduces certain randomness into the test process and that can be useful for uncovering certain sequences of actions which may have a negative impact of the performance. Such sequences would be difficult or maybe impossible to discover if static test scripts are used, where a fixed order of the actions is specified, and repeated over and over again. Every PTA has an exit location which will eventually be reached.

3.2.2 Creation of Workload Models

We describe two different ways of creating workload models. First we present a *systematic manual approach for creating workload models from scratch.* Second, we describe an *automatic approach for inferring workload models from historical data.* Using models, allows us to *hide implementation details and, instead, focus on relevant parts of describing user behavior.* In performance testing, it is important that the load generated from workload models *mimic the load generated by real users as closely as possible*, otherwise it is not possible to draw any reliable conclusions from the test results [23]. Having automatic and systematic approaches for creating workload models *ensure that our models stay close to real user behavior.*

Systematic Creation of Workload Models

Our first proposal is a *systematic approach for manually creating workload.* Having a systematic approach is beneficial because it is *repeatable* and it also tells one *how and where to obtain the necessary information.* The first step in manually creating workload models is characterizing the workload of the system. Menasce and Almeida [56] state that the workload of a system can be defined as the set of all inputs the system receives from the environment during any given period of time. In modern systems, it is virtually impossible to construct probabilistic models that fully describe all possible interactions with a system, due to state space explosion. However, if possible, such models would still be difficult for humans to comprehend and certainly challenging to maintain. Hence, the workload of a system can be seen as a set of key performance scenarios. This idea is based on the fact that certain scenarios are more frequent than others and that some impact more on the performance of the system than other scenarios.

We start by analyzing the requirements, Service Level Agreements (SLAs), and the system specifications, respectively. By using these sources we identify the inputs of the system with respect to types of transactions, transferred files, arrival rates, etc., following the generic guidelines discussed in [57]. Secondly, we try to form an understanding of which are the different types of users and how they interact with the system. Finally, we identify what are the key performance scenarios for each user type that will impact most on the performance of the system. A user type can be seen as a set of users that share a common behavior and is characterized by the distribution and the types of actions it performs. Each identified user type is represented in the user profile and has a separate workload model that describes the probabilistic behavior. In addition, we extract information regarding the KPIs, such as the number of concurrent users the system should support, expected throughput, response times, expected resource utilization demands etc. for different actions under a given load. We would like to point out that this is a manual step in the process. The results of the workload characterization are aggregated in a workload model similar to the one in Figure 3.2

Automatic Creation of Workload Models

A second way of constructing a workload model is to *use historical log data as a source of information* [58]. We propose an *automated approach that infers a set of workload models from web server log data*. This approach is beneficial because it is *less prone to errors*, *significantly reduces model creation time*, and *maps better to real user behavior* compared to manual approaches. The starting point of this approach is a web server log provided by web servers such as Apache [59] or Microsoft Server [60]. A server log is a list of entries that describe requests for different types of resources. The entries in the log contain detail information about the requests, such as, IP-address, the time of request, the request method, the request resource, the status code, etc. Table 3.1 shows an example of a typical logging format.

IP-address	User-Identifier	User Id	Date	Requested Resource	Status	Size
87.153.57.43	example.com	bob	[20/Aug/2014:14:22:35 -0500]	"GET /browse HTTP/1.0"	200	855
136.242.54.78	example.com	alice	[20/Aug/2014:24:22:45 -0700]	"GET /browse HTTP/1.0"	200	855
87.153.57.43	example.com	bob	[20/Aug/2013:14:22:56 -0500]	"GET /basket/book_42/add HTTP/1.0"	200	685
136.242.54.78	example.com	alice	[20/Aug/2014:14:23:04 -0700]	"GET /basket/phone_6/add HTTP/1.0"	200	685
87.153.57.43	example.com	bob	[20/Aug/2013:14:23:58 -0500]	"GET /basket/book_42/delete HTTP/1.0"	200	936
136.242.54.78	example.com	alice	[21/Aug/2014:14:54:02 -0700]	"GET /basket/view.html HTTP/1.0"	200	1772

Table 3.1: Example of a typical web server format

By analyzing the server log it is possible to deduce the request pattern of individual users. In our approach, we analyze and process the server log in several steps in order to produce workload model. First, we parse the log file to extract the the information for the individual entries. During this

step, requests from autonomous machines, also referred to as bots, are ignored since they are considered as irrelevant candidates for key performance scenarios. Users are identified with their IP-address and requests made from the same users are kept is separate lists. Secondly, we split the list of request of each user into shorter list, called *sessions*, based on a predefined session time-out value. A session is a sequence of requests to the web server which represent the user activity in a certain time interval from the same user.

In the next step, we are trying to deduce user actions from a set of web requests. As stated earlier in this chapter, actions can be seen as abstract transactions or templates that fit many different requests. Actions can be quite similar in structure, yet, not identical to each other. For example, consider a normal web shop where users add products to the basket. Adding two different products to the basket will result in two different web requests even though the underlying user action is the same. To achieve this, we structure the request in a tree-like manner and, later, reducing the tree by grouping together nodes that share joint sub-nodes. Once the tree has been reduced to a minimum, every path leading to a leaf node is considered as an action.

We then classify users based on their request patterns using the K-means algorithm [61]. Users with a distinctly different access pattern are clustered in separate groups. This is done in order to obtain a separate workload models for user types with distinctly different behaviors. Before constructing workload models for each identified group of users, we filter out sessions with a low frequency according to a Pareto probability density function [62] by cutting off the tail beneath a certain threshold value. Sessions with a low frequency do not impact significantly on the system's performance and can thus be neglected. Including all sessions would result in a workload model that is too cluttered and difficult to understand and maintain.

In a step-wise manner we then build a workload model where we overlap the remaining sessions of all users belonging to the same cluster. Session by session we gradually build a model, while reusing existing nodes in the model as much as possible. In order to calculate the probability and an average think time value for each edge we keep track of the number of times each edge has been reused.

Requirements Traceability in MBPeT

In our research, we also *trace non-functional requirements across the model-based performance testing process.* This allows us to *compare the measured KPI values against target values set prior to testing.* Target response time values are defined for individual action and monitored throughout the testing process. Whenever a target level is reached, the MBPeT tool reports on the current number of concurrent users and time of the breach. Figure 3.4 shows

a table where target response time values have been defined for individual actions. For example, for every action, an average and maximum threshold value is defined.

AVERAGE/MAX RESPONSE TIME THRESHOLD BREACH per METHOD CALL

Action	Target Response Time		NON-BIDDER_USER		PASSIVE_USER		AGGRESSIVE_USER		Verdict
	Average (secs)	Max (secs)	Average users (secs)	Max users (secs)	Average users (secs)	Max users (secs)	Average users (secs)	Max users (secs)	Pass/Fail
GET_AUCTION(ID)	2.0	4.0	70 (251)	84 (299.0)	70 (251)	95 (341.0)	70 (250)	95 (341.0)	Failed
BROWSE()	4.0	8.0	84 (299)	97 (345.0)	84 (299)	113 (403.0)	84 (299)	113 (403.0)	Failed
GET_BIDS(ID)	3.0	6.0	84 (298)	112 (402.0)	83 (296)	112 (402.0)	96 (344)	112 (401.0)	Failed
BID(ID,PRICE,USERNAME,PASSWORD)	5.0	10	Passed	Passed	97 (346)	113 (405.0)	112 (402)	135 (483.0)	Failed
SEARCH(STRING)	3.0	6	95 (341)	134 (479.0)	96 (342)	112 (402.0)	83 (296)	133 (476.0)	Failed

Figure 3.4: Table showing traceability of response time values to actions/method calls

After each test run, the *measured* average and maximum will be displayed together with the *target* values. If any of the target values have been breached, the tool reports how long into the test run the threshold was breached and how many concurrent users the tool was running at that point. If the target value was not breached, the tool marks it with a pass. For example, in figure 3.4 we see that the target average response time value of 2 seconds for the *get_auction* action for the *aggressive user* type was breached 250 seconds into the test run when running with 70 concurrent users. From this information we can conclude that; if the system must guarantee an average response time of 2 seconds for the *get_auction* action, given that a third of the users are of the type *aggressive users*, then the system cannot support more than 70 concurrent users. Besides just measuring response time values, the tool also monitors throughput, CPU, memory, disk, and network utilization.

3.3 Validation

In this section we describe how each contribution has been validated. We present tool support and empirical evaluation of the presented research performed on case studies

3.3.1 Tool Support

Support for load generation from workload models

MBPeT [63] [64] is a Python-based tool that generates the load using a distributed architecture and applies it in real-time to the system under test. During the load generation process, the tool measures several key performance indicators (KPIs), such as response time, throughput, error rate,

etc. After each test session, a test report is generated containing aggregated information about the each measured KPI.

Architecture. The architecture of the MBPeT tool [63] has been designed to fit in a cloud environment. It supports load generation over multiple machines in a distributed fashion. The tool architecture is made up of two types of nodes: a master node and slave nodes. A single master node controls multiple slave nodes, as shown in Figure 3.5. The slave nodes can be distributed, meaning that their physical location is not dependent on the location of the mater node. Slave nodes are designed to be homogeneous and generic, in the sense that they do not have prior knowledge of the SUT, its interfaces, or the workload models. Hence, the master node is responsible for providing all the necessary information required for load generation.

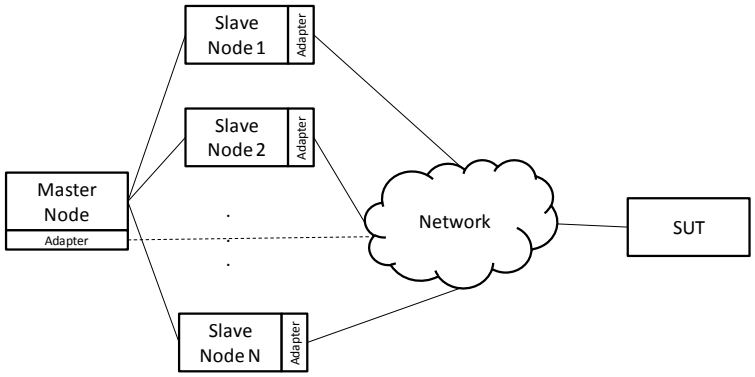


Figure 3.5: Distributed architecture of MBPeT tool

Load Generation. Load generation is initiated by activating a single slave node. New generation nodes are allocated dynamically during load generation as more resources are needed. Load is generated from PTA models by creating traces from the corresponding PTA. Creating traces is a step-wise process and is based on a deterministic choice with a probabilistic policy at each step. This introduces a certain randomness into the load generation process and is good for mimicking the dynamic behavior of real users. Such sequences would be difficult or maybe impossible to discover if static test scripts are used, where a fixed order of the actions is specified, and repeated over and over again. By executing several workload models in parallel, we can simulate concurrent user behavior. The amount of load that is generated is dependent on the amount of concurrent models that are executed in parallel. The idea is that one instance of a workload model corresponds to one user. The number of desired users at each time instance can be describe with a *ramp* function. The ramp function can be seen as a spline that is piecewise-defined by polynomial functions.

Throughout the whole load generation process, the target KPIs values

were constantly monitored. At the end, we collect all the gathered data and compute descriptive statistics [63]. For example, figure 3.6 shows a graph of how the throughput varied during the test run.

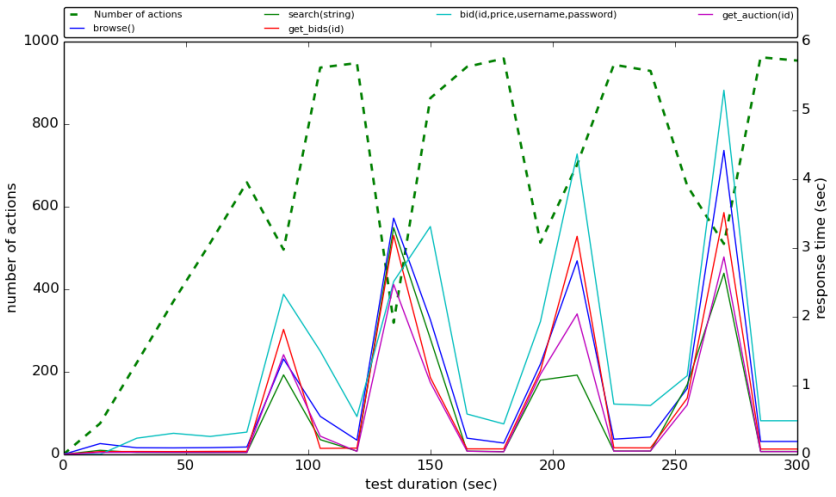


Figure 3.6: Throughput of the course of a test run.

Test Reporting. After each test run the MBPeT tool generates a test report based on the monitored data. All the gathered information is presented in a test report. The resource utilization of the SUT is also monitored and reported. Besides computing different kinds of statistical values from the raw data we have, the test report also contains graphs such as how the response time varied over time with the number of concurrent users. The test report also shows the CPU, disk, network and memory usage on the target system [65]. A complete description of the tool and test reporting is presented in [64].

Log2Model Tool

Log2Model is a Python-based tool [58] for generating workload models from server logs. The tool has a set of pre-defined patterns for common logging formats that are typically used in modern web servers (e.g., Apache and Microsoft Server). However, if a custom logging format is used, it is possible to manually specify the logging format via a regular expression. Parsed log files are stored in a database, minimizing the effort of the algorithm having to re-parse large log files between experiments.

One of the main features of the tool, apart from generating a workload model, is the user classification. This option separates users with distinctly different request pattern into separate groups. Another nice feature is that

once the workload models have been generated, the user is left with the choice of adjusting the cut-off threshold value, which determines how many sessions to include in the model (see Figure 3.7). This way the user can set a desired level of complexity for the generated models. This is done without having to re-run the algorithm, instead only the model is re-drawn based on the selected value.

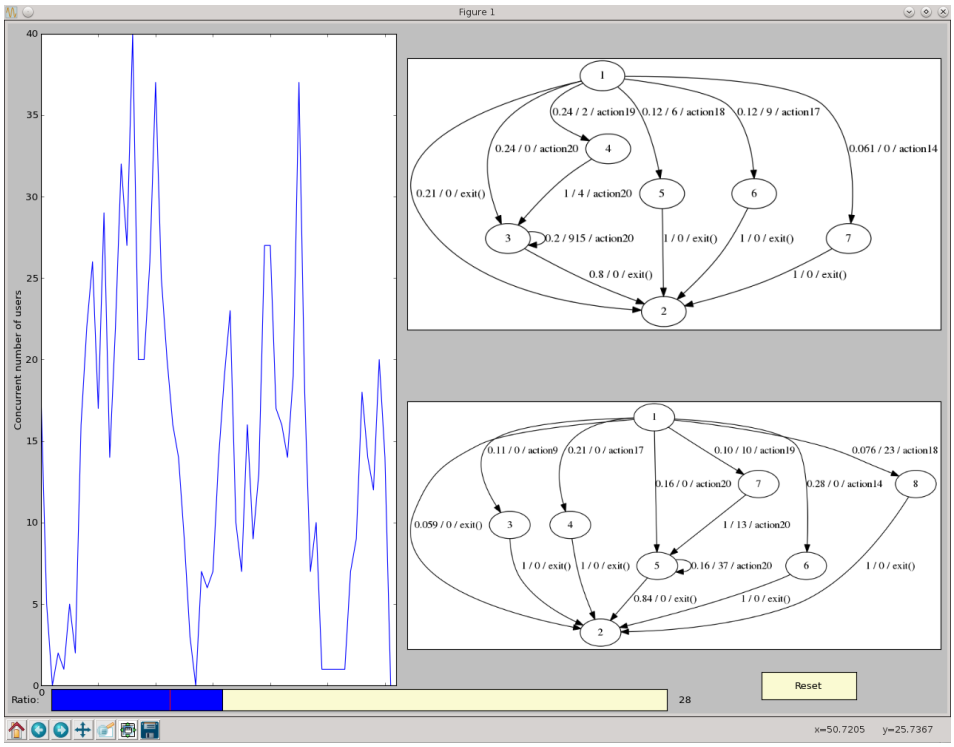


Figure 3.7: Screenshot of the GUI of the Log2Model tool

When a desired complexity of the workload model is chosen, the model is saved. Upon saving the models another artifact is created, namely a Python adapter code. The adapter code contains the mapping of each action in the models in a parameterized form and is used to interface our MBPeT tool with the system under test.

3.3.2 Empirical Validation on Case Studies

In this chapter, we will show our model-based performance testing approach [58] [63] in practice and demonstrate its applicability with a set of experiments on a case study. We show that using abstract models for describing the user profiles allows us quickly to experiment with different load mixes and detect worst case scenarios. We will also show how using log data as a

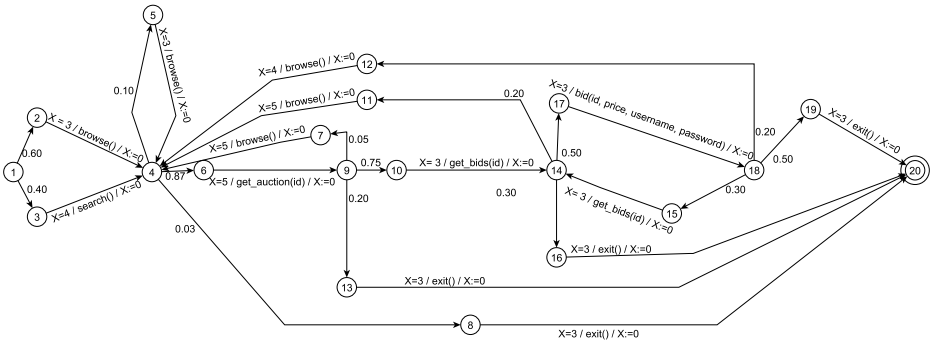


Figure 3.8: PTA model for an *aggressive-bidder* user type.

basis for workload model creation can radically reduce the effort needed to create workload models.

Generating Load With MBPeT

We demonstrate the applicability of our tool and approach by using it to evaluate the performance of an auction web service, generically called YAAS. The web service was developed in-house and implemented as a stand-alone application. The YAAS has a RESTful [66] interface based on the HTTP protocol and works as a normal auctioning web site where users create, search, browse, and bid on items that other users have created.

In the experiment we created three types of users: Passive, Aggressive, and Non-Bidders. The idea was that Aggressive users bid frequently on items while Passive users bid less on items. Non-bidders do not bid on items at all but only search and browse through the catalog of items. Figure 3.8 shows the PTA model for an aggressive user. Similar models were created for the two other user types.

We set out to test how many concurrent users the host node can support without exceeding the specified target response time values that we had set as thresholds for different actions. The length of the test was 20 minutes and the number of concurrent users were linearly ramped up from 0 to 300 over the course of the test run. From table 3.2 it can be seen that the system could support 64 concurrent users before breaching one of the set response time threshold values. The MBPeT tool also maintains a log file of all the traces generated from the workload models during each test run. In this particular example, the tool generated a total of 2576 unique traces in 20 minutes. The length of the traces varied from 2 up to 45 actions.

	Target Response Time		Non-Bidders (22 %)		Passive Users (33 %)		Aggressive users 45 %		Verdict
Actions	Average (sec)	Max (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Pass/fail
browse()	4.0	8.0	279 (78 users)	394 (110 users)	323 (90 users)	394 (110 users)	279 (78 users)	394 (110 users)	Failed
search(string)	3.0	6.0	279 (78 users)	394 (110 users)	279 (78 users)	394 (110 users)	229 (64 users)	327 (92 users)	Failed
get_action(id)	2.0	4.0	280 (79 users)	325 (91 users)	279 (78 users)	279 (78 users)	276 (77 users)	325 (91 users)	Failed
get_bids(id)	3.0	6.0	279 (78 users)	446 (130 users)	325 (91 users)	394 (110 users)	327 (92 users)	394 (110 users)	Failed
bid(id,price, username, password)	5.0	10.0	—	—	327 (92 users)	474 (132 users)	328 (92 users)	468 (131 users)	Failed

Table 3.2: Response time measurements for user actions when ramping up from 0 to 300 users.

Traces	JMeter		MBPeT	
	Nr of Traces	Percent of Traces	Nr of Traces	Percent of Traces
browse,get_auction	3706	27,77 %	3682	27,31 %
search,get_auction	3101	23,24 %	3218	22,87 %
browse,get_auction,get_bids	2427	18,19 %	2447	18,15 %
search,get_auction,get_bids	2306	17,28 %	2305	17,10 %
browse,get_auction,get_bids,bid	1803	13,51 %	1831	13,58 %
total	13343	100%	13483	100%

Table 3.3: JMeter vs MBPeT: Running 5 different traces with 100 users in parallel for 20 minutes. A uniform think time of 3 seconds between actions was used

Comparing MBPeT to JMeter

We investigated how our approach compares to JMeter [67], a well-know java-based load generator. In this experiment, we ran 5 different test sequences with 100 concurrent users and a test session of 20 minutes with a ramp-up period of 120 seconds. Between each action a uniform *think time* of 3 seconds was used. The tests were run three times and an average was computed. The test sequences used in this experiment were selected based on a previous test run of the YAAS application, where the top 5 most executed sequences were selected for this experiment. We constructed a model containing the 5 selected trace in MBPeT and did the same in JMeter.

The test sequences and the distribution between them can be seen in Table 3.3. The table shows that the JMeter tested on average a total of 13343 test sequences while the MBPeT tested on average a total of 13483 test sequences. This corresponds to a 1 percent speed advantage for the MBPeT tool. We note that there is a difference in the percentages in which test sequences were executed. This is because the MBPeT tool uses probabilistic models, from which load is generated, and due to the randomness in the models it is difficult to control the exact distribution between test sequences.

The test was not conducted to show that our tool in any way is better than JMeter, but rather to show that our tool can compete with modern load generator in terms of generation speed. In this experiment it did require less time to develop the JMeter scripts than it took to make our models and adapter. This is due to the fact that JMeter is a mature load generator that comes equipped with all kinds of ready-made templates, databases, and other elements which our tool lacks. However, because we are using

graphical models, our approach allows for a higher degree of model reuse between experiments compared to JMeter.

Log2Model

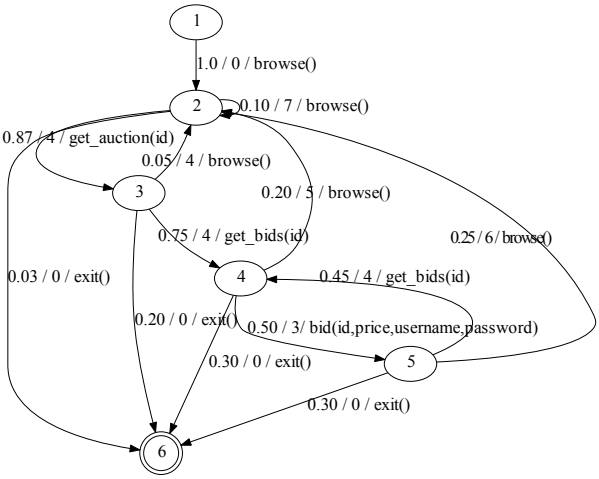
To validate the applicability of our Log2Model tool we tried to re-create the workload models from log data that was produced during an earlier test run of the YAAS application. We created a set of PTA models for the YAAS system and loaded them in our MBPeT tool. We ran load tests based on the models for 2 hours in order to produce a log file. From the produced log file, containing roughly 10,000 requests, we tried to re-create the original models as accurately as possible using our Log2Model tool.

Figure 3.9 shows a comparison between the original model and the re-created model. From this figure one can see that the two models are virtually identical apart from some probability values on a few edges. This is due to the fact that we use a stochastic model for generating the load and we do not have an exact control over what traces are generated. With a larger log file, approximately 1,000,000 lines, the probability values would be expected to be even closer to the original values, if not exact. The paper shows that automation can reduce the effort necessary for creating workload models for performance testing. In contrast, Cai et al. [68] report that they spent around 18 hours manually creating a test plan and the JMeter scripts for the reference Java PetStore application [69]. In our experiment, parsing the 10,000 line took 2 seconds, while the rest of the information processing, including creating the models took less than a second. Table 3.4 shows a summary of the execution time for different steps of the algorithm for different log sizes. The final step, constructing the workload model, was purposely left out since it varies considerably depending on the chosen number of clusters and threshold value.

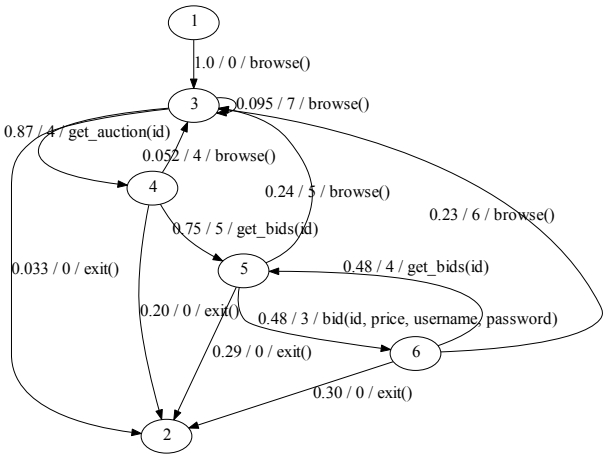
Phase	30.000	50.000	100.000	200.000	400.000
Parsing	6 sec	10 sec	22 sec	50 sec	2 min
Pre-processing	4 sec	9 sec	10 sec	21 sec	31 sec
Request tree reduction	0.3 sec	0.3 sec	0.8 sec	2 sec	5 sec
Clustering	0.08 sec	0.08 sec	0.4 sec	5 sec	60 sec
Total	10.38 sec	19.38 sec	33.2 sec	1 min 18 sec	3 min 36 sec

Table 3.4: Table showing execution times for different log sizes in terms of lines of log entries.

In a second experiment we show how complex models can be generated from only a small amount of log data. In this experiment we used log data from a web site called *pubiliiga* [70]. The web site maintains football scores in the league of pubiliiga. The web site also stores information about



(a) Original model.



(b) Recreated model.

Figure 3.9: Original and re-created workload models.

where and when the games are played, rules, game scores, teams, etc. The web site has been created using the Django framework [71] and runs on top of an Apache [59] web server. The results that we are going to show in this section are generated from a mere 30,000 lines of log data (7 MB) out of the original 1.3 million lines (320 MB). Generating the model on a computer equipped with an 8 core Intel i7 2.93 GHz processor and 16 GB of memory took approximately 2 seconds. After a model has been generated, the user decides the desired level of complexity on the model. This is done by choosing a threshold value. Traces are ordered according to their execution frequency as they are found in the log. Some traces are executed more than

others. For example, a threshold value of 20 percent (0.2) means that a model is constructed with 20 percent of the most frequent traces and the rest will be filtered out. Figure 3.10 shows the generated workload model when using a threshold value of 0.5, which means that 50 percent of the traces are included in the model, starting from the high-frequent ones.

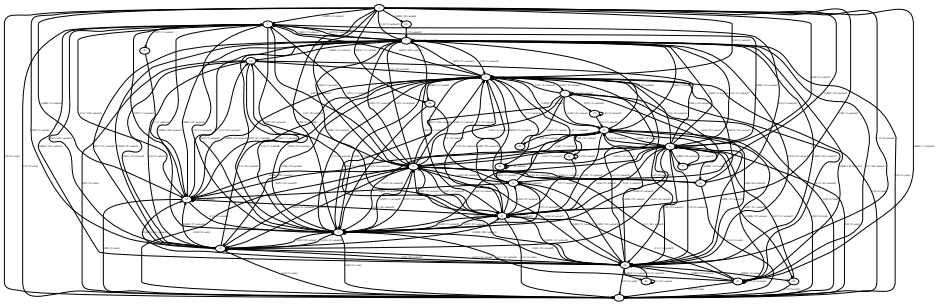


Figure 3.10: Models created from *pubiliiga.fi* log data. Threshold set at 50 percent

This experiment serves to show that complex models can be generated even when using relatively small amounts of data and when many traces are filtered out. From this experiment we can conclude that generating a model that perfectly matches reality would be highly complex and unmaintainable. But moreover, having a performance test where only a few static traces are repeated over and over again would be a huge simplification of reality.

3.4 Related Work

The related work presented in this section is divided into two parts; the first part addresses research work related to different model-based performance testing approaches while the second part presents work related to different performance testing tools. The approaches presented below is not an exhaustive list of related work but they are the ones that we find the most interesting or similar to our approach.

3.4.1 Performance Testing Approaches

Traore et al., [72] propose an SPE process using the UML profile for Schedulability, Performance and Time where they annotate UML models with performance data. However, no tests are derived from the models but instead the the annotation can be used for performance predictions. Similar approaches, where the performance of the system is simulated or predicted, are also described in [[73],[74], [75], [76], [77], [78]]. An approach using Petri Nets to evaluate performance of UML design is presented in [79]

Barna et al., [80] present a model-based testing approach to test the performance of a transactional system. The authors make use of an iterative approach to find the workload stress vectors of a system. An adaptive framework will then drive the system along these stress vectors until a performance stress goal is reached. They use a system model, represented as a two-layered queuing network, and they use analytical techniques to find a workload mix that will saturate a specific system resource. Their approach differs from ours in the sense that they use a model of the system instead of testing against a real implementation of a system.

Other related approaches can be found in [81] and [82]. In the former, the authors have focused on generating valid traces or a synthetic workload for inter-dependent requests typically found in sessions when using web applications. They describe an application model that captures the dependencies for such systems by using Extended Finite State Machines (EFSMs). Combined with a workload model that describes session inter-arrival rates and parameter distributions, their tool SWAT outputs valid session traces that are executed using a modified version of httperf [83]. The main use of the tool is to perform a sensitivity analysis on the system when different parameters in the workload are changed, e.g., session length, distribution, think time, etc. In the latter, the authors suggest a tool that generates representative user behavior traces from a set of Customer Behavior Model Graphs (CBMG). The CBMG are obtained from execution logs of the system and they use a modified version of the httperf utility to generate the traffic from their traces. The methods differ from our approach in the sense they both focus on the trace generation and let other tools take care of generating the load/traffic for the system, while we do on-the-fly load generation from our models.

Denaro [84] proposes an approach for early performance testing of distributed software when the software is built using middleware components technologies, such as J2EE or CORBA. Most of the overall performance of such a system is determined by the use and configuration of the middleware (e.g. databases). They also note that the coupling between the middleware and the application architecture determines the actual performance. Based on architectural designs of an application the authors can derive application specific performance tests that can be executed on the early available middleware platform that is used to build the application with. This approach differs from ours in that the authors mainly target distributed systems and testing of the performance of middleware components.

3.4.2 Performance Testing Tools

There exist a plethora of commercial performance testing tools. In the following, we briefly enumerate a couple of popular performance testing tools.

FABAN is an open source framework for developing and running multi-tier server benchmarks [85]. FABAN has a distributed architecture meaning load can be generated from multiple machines. The tool has three main components: A harness - for automating the process of a benchmark run and providing a container for the benchmark driver code, a Driver framework - provides an API for people to develop load drivers, and an Analysis tool - to provide comprehensive analysis of the data gathers for a test. Load is generated by running multiple scripts in parallel.

JMeter [67] is an open source tool for load testing and measuring performance, with the focus on web applications. JMeter can be set up in a distributed fashion and load is generated from manually created scenarios that are run in parallel. Httperf [86] is a tool for measuring the performance of web servers. Its aim is to facilitate the construction of both micro and macro-level benchmarks. Httperf can be set up to run on multiple machines and load is generated from pre-defined scripts.

LoadRunner [87] is a performance testing tool from Hewlett-Packard for examining system behavior and performance. The tool can be run in a distributed fashion and load is generated from pre-recorded scenarios. Recently several authors have focused on using models for performance analysis and estimation, as well as for load generation.

Little research has been published in the area of generation of workload models using log data. However, when is comes to log data, some research exists in user clustering and pattern detection.

Kathuria et al. proposed an approach for clustering users into groups based on the intent of the web query or the search string [88]. The proposed approach clusters web queries into one of the three categories based on a K-means algorithm. Vaarandi [89] proposes a Simple Logfile Clustering Tool consequently called SLCT. SLCT uses a clustering algorithm that detects frequent patterns in system event logs. The event logs typically contain log data in various formats from a wide range of devices, such as printers, scanners, routers, etc. The approach is using data mining and clustering techniques to detect normal and anomalous log file lines. Shi [90] presents an approach for clustering user's interest in web pages using the K-means algorithm. The approach uses fuzzy linguistic variables that describe the time duration that users spend on web pages as the classification variable. These approaches are different from ours in the sense that we assume to know the logging format and we classify users into group based on the request pattern rather that the intent, meaning, or time spent of a web page.

The solutions proposed by Mannila et al. [91] and Ma and Hellerstein [92] are targeted towards discovering temporal patterns from event logs using data mining techniques and various association rules. Both approaches assume a common logging format. Although association rules algorithms are powerful in detecting temporal associations between events, they do not

focus on user classification and workload modeling for performance testing. Another approach is presented by Anastasiou and Knottenbelt [93]. Here, the authors propose a tool, PEPPERCORN, that will infer a performance model from a set of log files containing raw location tracking traces. From the data, the tool will automatically create a Petri Net Performance Model (PNPM). The resulting PNPM is used to make an analysis of the system performance, identify bottlenecks, and to compute end-to-end response times by simulating the model. The approach differs from ours in the sense that it operates on different structured data and that the resulting Petri Net model is used for making a performance analysis of the system and not for load generation. In addition, we construct probabilistic time automata (PTA) model from which we later on generate synthetic load.

Lutteroth and Weber describe a performance testing process similar to ours [94]. Load is generated from a stochastic model represented by a form chart. The main differences between their and our approach is that we use different types of models and that we automatically infer our models from log data while they create the models manually.

3.5 Conclusions

In this chapter, we have presented our modeling approach for modeling for performance testing. We have presented the main contributions and explained how they have been validated. We have also shown how our research has been validated in the terms of the tools that have been built and the empirical validation that was made in different experiments. For example, we showed how modeling can raise the abstraction level and hide implementation details. We have also showed that fairly complex models can been generated in a short amount time from small amounts of data and that large amount our unique traces can be generated from very simple models. These results together serve as a basis for answering the questions of how create models from load generation and how to ensure that load generated from the models stays close the load generated by real users.

Chapter 4

Conclusions

In this thesis we have presented an approach for using models for test generation. The presented approach target both functional testing as well as performance testing. We have also sought solutions to the problems addressed Chapter in 1, i.e. that most model-based testing approaches are not integrated with a software development process and that most MBT approaches lack empirical evaluation for an industrial environment. Also, a considerable amount of the reviewed literature lack or only offer partial tool support. To address these issues, we offer tool support that are integrated with well-known software development tools or work as stand-alone tools. We also demonstrate the applicability of our approach with empirical evaluations taken from the industry.

For model-based functional testing, we presented the MATERA approach. The MATERA approach include several steps: model creation, model transformation, test generation, test execution. On top of this, requirements are traced through all of these steps. The main contribution is the actual modeling approach. Here, we show how the SUT models are built in a stepwise manner and how requirements and statistical information are included and traced to different model elements. This allowed requirement information to be propagated to test generation tools and be used as test generation goals and even test case ordering. Another important contribution is tool support. We presented the MATERA tool-set, and showed how it integrates with MagicDraw, one of the leading UML tools on the market. We discussed how the tool aid the tester in different phases, such as, model validation, model transformation, and requirements traceability. Finally, we demonstrated the applicability of our approach and tool support by validating it against an industrial case study. The goal of the case study was to investigate how to apply MBT in an industrial setting. The system under test was a mobile switching server. The main features of the mobile switching server that we investigated was the *location update*, *voice call*, and

handover procedures. In total, 40 different UML diagrams were created in order describe the static and dynamic parts the SUT. The UML diagrams contained, among other things, 35 requirements, 5 use cases, 9 sequence diagrams, 88 states, 136 transitions, and over a 100 different classes. This information was then translated into QML, using the MATERA transformation module. From the resulting QML models, a total of 114 test cases were generated. The generated test cases reveled no errors. This was mainly because, in this case study, the generated test cases was executed on production Mobile Switch Server (MSS) where no errors were expected to be present.

To address the problems of model-based performance testing, we introduced MBPeT. MBPeT is a load testing process with assisted tool support. We described how our load testing process make use of workload models described as probabilistic timed automata (PTA). Due to their probabilistic nature, PTA models are suitable for representing user behavior. One of the main contribution is the process of creating workload models. We presented three different approaches for creating workload models: (a) automatic derivation of PTA models from UML sequence charts, (b) automatic derivation of PTA models from log data, (c) manual systematic creation of PTA models from system specifications. Another important contribution is our process-assisted tool support. Here, we presented the MBPeT performance testing tool. MBPeT is a cloud-based performance testing tool that generates load using is distributed architecture. During load generation, the tool measure several key performance indicators, such as, response time, CPU utilization, memory usage, etc. At the end of the test run, the tool also presents a test report with the aggregated data.

Finally, we showed how MBPeT can be used in an industrial setting. We demonstrated the applicability of the tool on an auctioning web service and made a comparison with the well known-load generator JMeter. The comparison showed that MBPeT is on par with JMeter and even sometimes faster. Since the comparison was made, several improvements have been done to MBPeT. We also demonstrated the applicability of the tool support for creating workload models from log data. The experiments showed that PTA models can be generated in a matter of minutes, or even seconds. This is a huge speed-up compared to other manual approaches that requires up many hours in order to produce a workload model that accurately describes user behavior.

4.1 Discussion

In the next section, we acknowledge the limitations in the work presented in this thesis.

One of the most difficult problems to tackle in design science and research in general is the ability to generalize the results. In our research work, we acknowledge the fact that the results of our experiments and finding has been derived from one single case study where only one test system was used. To be able to further generalize our results, our approach would need to be applied on other systems in other domains. However, in an academic setting, developing a methodology and testing it thoroughly, simply lies beyond the capacity and focus of this thesis. Further, obtaining additional interesting and complex industrial case studies for verifying the results of our proposed approach also require the willingness of industrial partner to collaborate in a desirable fashion. We recognize that this is not always easy to achieve. We also recognize that our experiments, in the context of model-based functional testing and performance testing, were performed on different systems under test. Ideally the same test system should have been used. The reason for this is that the two approaches were developed in different stages and different projects were different industrial partners had their own needs and systems.

The work done in the context of model-based functional testing and MATERA was attributed to one particular tool-chain. We note the fact that in order to be able to fully generalize our results, other test generation and execution tools should also be targeted. However, in the context of our work, we have tried to target and integrate our work with leading tools for each part in the testing process. During the development of the process, a lot of effort was put into tool integration. Despite the fact that the models were described using standardized modeling languages, such as UML and SysML, it was not possible to create a UML model using one tool and open it with another tool. Transforming the models between the tools can be used to circumnavigate this problem. Consequently, transformations will be performed frequently, and hence there would be a tool-chain induced performance penalty. In addition, implementing a number of model transformations increases the costs of the tool-chain. The tools used in our tool-chain were all commercial and industry leading tools. Creating adaptation for an entire new tool-chain was simply out of the scope for this thesis.

The MBPeT tool described in chapter 3 has been developed and tested in a private cloud with only 4 computing nodes. Public clouds, unlike private clouds, are usually preferred by organisations due to their availability and on-demand scalability. Due to the choice of targeting a private cloud, the MBPeT tool has not yet been tested to its limits. For a wider adoption of our research work regarding model-based performance testing, we acknowledge

the fact that we would need to target public clouds as well, for example Amazon EC2 [95]. However, we are limited to only certain clouds, due to that choice of using Python [96] as the implementation language of the MBPeT tool.

The work in the context of model-based performance testing and MBPeT was validate only against http-based systems. To be able to generalize the results in the context of MBPeT, we should also target other protocols and other systems in other domains. We recognize the difficulty in finding collaboration parters with interesting test systems and performance requirements that meet our needs. For this reason, we decided to implement our own test system for use in our experiments.

4.2 Future work

The research work presented in this thesis has shown a coherent approach for creating and using models for functional and performance testing. There are of course several potential places of improvements in our approach and tool support. In the following, we mention a few improvements and some future research directions.

In the context of model-based functional testing, we have focused mainly on generating input for certain tools in our tool-chain. To get a better sense of the generalization of our work, it would certainly be interesting to investigate how our methodology translate to other test generation and execution tools used in the industry. Moreover, extending the approach to target other types of systems in other application domains would certainly be beneficial.

It is really difficult to get a good understanding of the weak points and the real potential of our approach without solid empirical data. Making experiment in a laboratory environment is simply not enough. It would certainly be interesting to let the industry try our methodology and tool support to get empirical data on how much time and effort that really is saved using our approach. This is something we hope to achieve in the future.

So far, the MBPeT tool has only been used for testing web services. However, the approach is not limited only to such systems. In the future, we plan to address web applications, as well as other types of communicating systems in other domains. Due to limited resources, the MBPeT tool has never been tested with extreme load levels. Hence, we do not yet known the upper capacity of the MBPeT tool. In the future, we plan to investigate this further to find out how much load the tool can handle.

Bibliography

[1] M. Haug, E.W. Olsen, and L. Consolini. *Software Quality Approaches: Testing, Verification, and Validation: Software quality approaches : testing, verification, and validation.* ESSI practitioners' reports. Springer Berlin Heidelberg, 2001.

[2] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[3] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[4] Matthew Heusser. *How to Reduce the Cost of Software Testing.* Auerbach Publications, 1 edition, September 2011.

[5] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality.* Addison-Wesley Professional, 1st edition, 2009.

[6] International Electrotechnical Commission. Iec 61508. `http://www.iec.ch/functionalsafety/standards/`.

[7] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[8] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[9] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2008.

[10] Sascha Kirstan Helmut Krcmar Broy, Manfred and Bernhard Schtz. What is the benefit of a model-based design of embedded software systems in the car industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 343–369. IGI Global, 2012.

[11] ITEA2. Itea 2 d-mint project result leaflet: Model-based testing cuts development costs. Online at: https://itea3.org/project/result/download/5519/D-MINT%20Project%20Leaflet.pdf, February 2015.

[12] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[13] Bernhard Peischl, Rudolf Ramler, Vitalina Turlo, Stefan Mohacsi, Valery Safronau, and Tobias Walter. Integrated model-based testing in a project's ttool landscape (white paper). Online at: http://nl.atos.net/content/dam/nl/documents/atos-wp-modelbasedtesting-tam.pdf, February 2015.

[14] Robert Binder. Model-based testing user survey: Results and analysis. Online at: http://robertvbinder.com/wp-content/uploads/rvb-pdf/arts/MBT-User-Survey.pdf, March 2015.

[15] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36. ACM, 2007.

[16] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, January 2002.

[17] Gregor Engels, Reiko Heckel, and Stefan Sauer. Uml – a universal modeling language? In D. Simpson M. Nielsen, editor, *In Proc. 21st International Conference in Application and Theory of Petri Nets*, volume 1825, pages 24–38, 2000.

[18] Laura A. Campbell, Betty H. C. Cheng, William E. McUmber, and R. E. K. Stirewalt. Automatically detecting and visualising errors in uml diagrams. *Requirements Engineering*, 7(4):264–287, 2002.

[19] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artif. Intell.*, 168(1):70–118, October 2005.

[20] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, 2004.

[21] Zoltan Micskei. Mbt tools. Online at: http://mit.bme.hu/ micskeiz/pages/modelbased_testing.html, March 2015.

[22] Terry Shepard. Incomplete list of tesing tools. Online at: http://research.cs.queensu.ca/ shepard/testing.dir/under.construction/tool_list.html, March 2015.

[23] J. Shaw. Web application performance testing — a case study of an on-line learning application. *BT Technology Journal*, 18(2):79–86, April 2000.

[24] Fredrik Abbors, Tuomas Pjrvi, Risto Teittinen, Dragos Truscan, and Johan Lilius. Transformational support for model-based tesing - from uml to qml. In *Proceddings of 2nd Workshop on Model-Based Testing in Practice (MOTIP'09)*, volume 1, pages 55–64, 2009.

[25] Deployment of model-based technologies to industrial testing (d-mint) project. Online at: https://itea3.org/project/d-mint.html, February 2015.

[26] Practical applications of model-based technologies to continuous integration & testing methodproject (pam) project. Online at. https://research.it.abo.fi/projects/PAM, February 2015.

[27] Doctoral programme on software and systems engineering (sose). Online at: http://www.sose.oulu.fi/, March 2015.

[28] Object Management Group (OMG). The unified modeling language. Web, August 2015. `http://www.omg.org/spec/UML/`.

[29] Object Management Group (OMG). The systems modeling language. Web, August 2015. `http://www.omg.org/spec/SysML/1.2/PDF/`.

[30] Object Management Group (OMG). Object constraint language specification formal/06-05-01. Web, February 2014. `http://www.omg.org/spec/OCL/`.

[31] NoMagic. Magicdraw user manual. Web, 2014. `http://www.nomagic.com/files/manuals/MagicDraw%20UserManual.pdf`.

[32] Conformiq. *Conformiq Designer 4.4 User Manual*, 2011. `http://www.verifysoft.com/ConformiqManual.pdf`.

[33] Nethawk. Nethawk east, 2008. `https://www.nethawk.fi/products/news/nethawk-and-testing-techn/index.xml`.

[34] Michael Schmitt, Michael Ebner, Technische Berichte, Georg august-universitt Gttingen, Michael Schmitt, and Michael Ebner. The ttcn3, 2003.

[35] Johan Abbors. Increasing Quality of UML Models Used for Automatic Test Generation. Master's thesis, Åbo Akademi University, 2009.

[36] Orlena Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, 1994.

[37] Tuomas Pääjärvi. Generation input for the test generator tool from uml design models. Master's thesis, Åbo Akademi University, 2009.

[38] 3GPP. 3gpp ts 23.002 v8.3.0 technical specification group services and systems aspects; network architecture, September 2008.

[39] 3GPP. 3gpp ts 44.018 v8.2.0 technical specification group services and systems aspects; network architecture, Mars 2008.

[40] Kim Nylund. From test generation to test execution in model-based testing. Master's thesis, Åbo Akademi University, 2010.

[41] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, volume 1. Addison Wesley Longman, Inc., 1999.

[42] F. Basanieri and A. Bertolino. A practical approach to uml-based derivation of integration tests. In *Proc. 4th International Software Quality Week Europe (QWE'2000)*, 2000.

[43] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[44] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 194–208, London, UK, UK, 2001. Springer-Verlag.

[45] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 86–95, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.

[47] Aynur Abdurazik and Jeff Offutt. Using uml collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd International Conference on The Unified Modeling Language: Advancing the Standard*, UML'00, pages 383–395, Berlin, Heidelberg, 2000. Springer-Verlag.

[48] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978.

[49] Avik Sinha and Amit Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '06, pages 17–22, New York, NY, USA, 2006. ACM.

[50] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[51] Alessandra Cavarra, Charles Crichton, and Jim Davies. A method for the automatic generation of test suites from object models. *Information Software Technology*, 46(5):309–314, 2004.

[52] Santosh Kumar Swain and Durga Prasad Mohapatra. Article:test case generation from behavioral uml models. *International Journal of Computer Applications*, 6(8):5–11, September 2010. Published By Foundation of Computer Science.

[53] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[54] Dorina C. Petriu and Hui Shen. Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. pages 159–177. Springer-Verlag, 2002.

[55] M. Jurdziński, M. Kwiatkowska, G. Norman, and A. Trivedi. Concavely-Priced Probabilistic Timed Automata. In M. Bravetti and G. Zavattaro, editors, *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*, volume 5710 of *LNCS*, pages 415–430. Springer, 2009.

[56] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: metrics, models, and methods.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[57] Maria Calzarossa, Luisa Massari, and Daniele Tessera. Workload Characterization Issues and Methodologies. In *Performance Evaluation: Origins and Directions*, pages 459–481, London, UK, UK, 2000. Springer-Verlag.

[58] Fredrik Abbors, Tanwir Ahmad, and Dragos Truscan. An automated approach for creatingworkload models from server log data. In *Proceedings of the 2014 9th International Conference on Software Engineering and Applications*, 2014.

[59] The Apache Software Foundation. Apache https server project. Online at: http://httpd.apache.org/, February 2015.

[60] Microsoft. Microsoft server 2012 r2. Online at: http://www.microsoft.com/en-us/server-cloud/products/windows-server-2012-r2/, February 2012.

[61] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, number 1, pages 281–297. Berkeley, University of California Press, 1967.

[62] BarryC. Arnold. Pareto and generalized pareto distributions. In Duangkamon Chotikapanich, editor, *Modeling Income Distributions and Lorenz Curves*, volume 5 of *Economic Studies in Equality, Social Exclusion and Well-Being*, pages 119–145. Springer New York, 2008.

[63] Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres. *Performance Testing in the Cloud using MBPeT*, volume 1, chapter 6, pages 191–226. Turku Centre for Computer Science, 2013.

[64] Tanwir Ahmad, Fredrik Abbors, Dragos Truscan, and Ivan Porres. Model-Based Performance Testing Using the MBPeT Tool. Technical Report 1066, Turku Centre for Computer Science (TUCS), 2013.

[65] Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres. Model-based performance testing of web services using probabilistic timed automata. In *Proceedings of the 2013 10th International Conference on Web Information Systems and Technologies*, 2013.

[66] Leonard Richardson and Sam Ruby. *Restful web services*. O'Reilly, first edition, 2007.

[67] The Apache Software Foundation. Apache JMeter. http://jmeter.apache.org/. Retrieved: October, 2012.

[68] Yuhong Cai, John Grundy, and John Hosking. Synthesizing client load models for performance engineering via web crawling. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 353–362. ACM, 2007.

[69] Oracle. Java Pet Store 2.0 reference application. http://www.oracle.com/technetwork/java/index-136650.html, 2015. Last Accessed: 2015-05-23.

[70] Kim Nylund. Pubiliiga.fi. Online at: http://www.pubiliiga.fi/, March 2015.

[71] Django. Online at https://www.djangoproject.com/, September 2012.

[72] Ahmed Awad El Sayed Ahmed Mohammed S. Obaidat Issa Traore, Isaac Woungang. Software performance modeling using the uml: a case study. *Journal of Networks*, 7(1):4–22, January 2012.

[73] Julie A. Street and Robert G. Pettit IV. Lessons learned applying performance modeling and analysis techniques. In *ISORC*, pages 208–214, 2006.

[74] V. Garousi. Uml model-driven detection of performance bottlenecks in concurrent real-time software. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2010 International Symposium on*, pages 317–324, July 2010.

[75] Vittorio Cortellessa, Antinisca Di Marco, Romina Eramo, Alfonso Pierantonio, and Catia Trubiani. Digging into uml models to remove performance antipatterns. In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, QUOVADIS '10, pages 9–16, New York, NY, USA, 2010. ACM.

[76] A. AL Abdullatif and R.J. Pooley. Uml-jmt: A tool for evaluating performance requirements. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 215–225, March 2010.

[77] Yong Zhang, Tao Huang, and Jun Wei. Declarative performance modeling for component-based system using uml profile for schedulability, performance and time. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 246–258, Sept 2006.

[78] Simonetta Balsamo and Moreno Marzolla. A simulation-based approach to software performance modeling. *SIGSOFT Softw. Eng. Notes*, 28(5):363–366, September 2003.

[79] Salvatore Distefano, Marco Scarpa, and Antonio Puliafito. From uml to petri nets: The pcm-based methodology. *IEEE Transactions on Software Engineering*, 37(1):65–79, 2011.

[80] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Model-based performance testing (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 872–875, New York, NY, USA, 2011. ACM.

[81] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. A model-based approach for testing the performance of web applications. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, pages 54–61, New York, NY, USA, 2006. ACM.

[82] G. Ruffo, R. Schifanella, M. Sereno, and R. Politi. WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance. *Network Computing and Applications, IEEE International Symposium on*, 0:77–86, 2004.

[83] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, December 1998.

[84] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th international workshop on Software and performance*, WOSP '04, pages 94–103, New York, NY, USA, 2004. ACM.

[85] Sun. Faban Harness and Benchmark Framework. http://java.net/projects/faban/, February 2013.

[86] Hewlett-Packard. Httperf. Online at: http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt, March 2015.

[87] HP. HP LoadRunner. http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/, February 2013.

[88] Ashish Kathuria, Bernard J. Jansen, Carolyn Theresa Hafernik, and Amanda Spink. Classifying the user intent of web queries using k-means clustering. In *Internet Research*, number 5, pages 563–581. Emerald Group Publishing, 2010.

[89] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations and Management (IPOM03)*, pages 119–126. IEEE, 2003.

[90] Peilin Shi. An efficient approach for clustering web access patterns from web logs. In *International Journal of Advanced Science and Technology*, volume 5, pages 1–14. SERSC, 2009.

[91] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, January 1997.

[92] Sheng Ma and Joseph L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of the 17th International Conference on Data Engineering*, pages 205–214, Washington, DC, USA, 2001. IEEE Computer Society.

[93] Nikolas Anastasiou and William Knottenbelt. Peppercorn: Inferring performance models from location tracking data. In *QEST*, Lecture Notes in Computer Science, pages 169–172. Springer, 2013.

[94] Christof Lutteroth and Gerald Weber. Modeling a realistic workload for performance testing. In *12th International Conference on Enterprise Distributed Object Computing.*, pages 149–158. IEEE Computer Society, 2008.

[95] Amazon elastic compute cloud 2. Online at http://aws.amazon.com/ec2/.

[96] Python. Python programming language. Online at http://www.python.org/. Last Accessed: 2014-12-30.

PART II

Original Publications

Paper I

Tracing Requirements In A Model-Based Testing Approach

Fredrik Abbors, Dragos Truscan, and Johan Lilius.

Originally published *2009 Proceeding of 1st International Conference on Advances in System Testing and Validation Lifecycle*. IEEE. September 2009, Porto, Portugal.

Tracing Requirements In A Model-Based Testing Approach

Fredrik Abbors, Dragoş Truşcan, and Johan Lilius,
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520, Turku, Finland
Email: {Fredrik.Abbors, Dragos.Truscan, Johan.Lilius}@abo.fi

Abstract

In this paper we discuss an approach for requirements traceability in a model-based testing process. We show how the informal requirements of the system under test evolve and are traced at different steps of the process. More specifically, we discuss how requirements are traced to system specifications and from system specification to tests during the test generation process, and then how the test results are analyzed and traced back the specification of the system. The approach allows us to have both a fast feed-back loop for debugging either the specification or the implementation of the system and a way to estimate the coverage degree of the generated tests with respect to requirements. We discuss tool support for the approach and exemplify with excerpts from a case study in the telecommunications domain.

1. Introduction

The key to successful product engineering in the software industry today is in many cases a good quality-assurance and deployment of software systems. Customers demand highly efficient, low-cost, and reliable software products. Companies are forced to build high-end products with a low budget in a short time. The increasing demand for software products forces the companies to develop new products at a very fast pace. We see a constant decrease in the time-to-market and customers demanding more flexible systems which result in the growing system complexity. Missing the deadline for the time-to-market can have a huge negative impact on the company's profit. Unfortunately, this fast pace leaves the companies with less time for testing their products.

The purpose of testing is to find faults that have been introduced during the development of the system, starting with the initial specification phases and ending with its implementation. Testing is also a means to ensure the quality of a product and to verify that the product meets its requirements. Having a systematic and automated way to test software systems would reduce the overall expenses of testing due to a shorter time-to-market. Likewise, this would leave the companies with more time for actually designing and implementing the software, and would result in better and more reliable software.

Model-based testing (MBT) is a software testing technique that has gained much interest in recent years by providing the degree of automation needed for shortening the time required for testing. The main idea behind MBT, is that a behavioral model of the system, namely a *test model*, is used for automatically deriving *test cases* following different coverage criteria.

The test model is typically developed from the informal requirements of the system and therefore it is important to trace how the generated test cases cover different requirements. Traceability of requirements can help one to achieve the right level of coverage and show what requirement has been covered by what test. Traceability can also be used to trace requirements to specifications (code or models) and can detect what part of a code or a model implements a requirement. By tracing requirements to tests, it becomes possible to trace back requirements to models, when a test fails and to identify from which part of a the test model the failure originated.

In this paper we present an approach for tracing product requirements across a model-based testing process, from informal documents via test models to test cases, and back to requirements and test models. The approach allows us to have both a fast feed-back loop for debugging either the specification or the implementation of the system and a way to estimate the coverage degree of the generated tests with respect to requirements. We discuss tool support for the approach and exemplify with excerpts from a case study in the telecommunications domain.

Related Work. Requirements traceability is a very popular topic in the software engineering and testing communities, and has gained momentum in the context of model-based testing in the context of automated test generation. However, as requirements change during the development life cycles of software systems, updating and managing traces has become a tedious task. Researchers have addressed this problem by developing methods for automatic generations of traceability relations [1] [2] [3] [4] by using information retrieval techniques to link artifacts together based on common key-words that occur in both the requirement description and in a set of searchable documents. Other approaches focus on annotating the model with requirements which are propagated through the test generation process in order to obtain a requirement traceability matrix [5]. The matrix is then used to manually analyze and track requirements to models. From the reviewed works, the one in [6] is closer to our approach. In there, the authors use textual delimiters to add requirements in the OCL constructs associated to a restricted set of UML models. The LEIROS test design tool is then used to generate test cases, and a traceability matrix is obtained after the test are executed. However, there is no tool support for tracing-back requirements from tests to models.

2. Model-Based Testing Process

Our model-based testing process (Figure 1) starts with the analysis and structuring of the informal requirements into a *Requirements Model*. The Requirements Diagrams of the Systems Modeling Language (SysML) [7] are used for this purpose. In the next phase, the system under test (SUT) is specified using the Unified Modeling Language (UML) [8]. In our modeling process, we consider that several perspectives of the SUT are required in order to enable a successful test derivation process later. In addition, one should note that in our approach a model of the system is used for deriving test cases and not a test model, the difference between the two being that the former is both used for development and testing, whereas the latter is only used for testing. Several perspectives of the SUT are modeled; a class diagram is used to specify a *domain model* showing what domain components exist and how they are interrelated through interfaces. A *behavioral model* describes the behavior of the SUT using state machines. *Data models* are used to describe the message types exchanged between different domain entities. Last but not least, *domain configuration* models are used to represent specific test configurations using object diagrams.

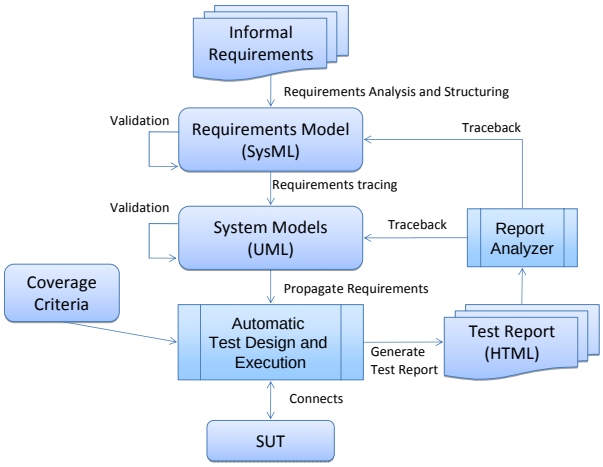


Figure 1: Overview of the model-based testing process

A set of modeling guidelines and validation rules are defined for ensuring the quality of the resulting models. Modeling guidelines are used to specify how different models are created from requirements or from other models, what information they should contain, how this information is related to the information present in the other models, etc.

Validation rules have been defined and implemented [9] for both Requirements Models and for System Models for checking different quality metrics of the resulting models before proceeding to the *test derivation phase*. These rules ensure that the models are syntactically correct, they are consistent with each other, and that they contain the information needed in the later phases of the testing process. Tool support is provided for automatically verifying these rules using the Object Constraint Language (OCL). The OCL rules check the static semantics of the models and can be used to describe constraints that are specific to the domain, modeling language, modeling process, etc. However, if OCL can be used for checking the dynamic semantics of the models has to be further investigated. The NoMagic's MagicDraw tool [10] has been used for editing the SysML and UML models and for running the validation rules.

The models used to specify the SUT are subsequently transformed into input for an automated test derivation tool, namely Conformiq's Qtronic [11]. We use the *online testing* mode of this tool, in which tests are generated and applied on-the-fly against the SUT. The desired coverage criteria used for test generation are manually selected from the graphical user interface (GUI) of Qtronic. At the end of each test run, a automatically generated *Test Report* will summarize

the result of the testing process in terms of generated test cases, verdicts, coverage levels, requirements traceability matrix, etc.

3. Requirements Traceability

Our approach to requirements traceability is built on top of the previously explained testing process with two goals in mind. Firstly, we want to be able to trace how different parts of the system models relate to the requirements and then to see how different requirements are covered by the generated test cases. Another reason for tracing requirements is that if a requirement changes, it is essential to know how this change is reflected in the models [12] [13]. Secondly, once the test report becomes available, we would like to be able to identify which requirements have been successfully tested and which have resulted in failures. In addition, for the failed test cases we should be able to trace back from test cases those parts of the SUT specification that generated the failure.

In the following, we briefly describe our requirements traceability approach while providing small examples from a telecommunications case study. In our case, the SUT is a Mobile Switching Server (MSS). The MSS is a network element located in a mobile telecommunication system. The MSS is connected to its surrounding elements through several different interfaces. The MSS is responsible for keeping track of the location of mobile subscribes (MS) in the network and for connecting calls between MS's over 2G and 3G networks. The MSS is also responsible for tracking the movement of MS's during an ongoing call.

3.1. Tracing requirements to tests

3.1.1. Requirements decomposition. The requirements models are obtained by analyzing informal requirements related to standards, protocols, system specifications, etc. Requirements are structured in a tree-like manner and defined on several levels of abstraction following a functional decomposition. They can also be related (i.e. traced) to other requirements on the same abstraction level. Requirements may also be decomposed into different categories, depending on the nature of the requirement, like functional, architectural, data, etc.

Each requirement element contains a *name* field which specifies the name of the requirement, an *id* field, and a *text* field. The *id* field simply specifies the id of the requirement, whereas the text field describes the requirement. A requirement also contains a *source* field. The source field specifies the origins of the requirement. The source can be a link to or a name of a textual document from where the requirement has been extracted. Figure 2 shows the functional requirements for the location update procedure of the MSS represented using a SysML requirements diagram.

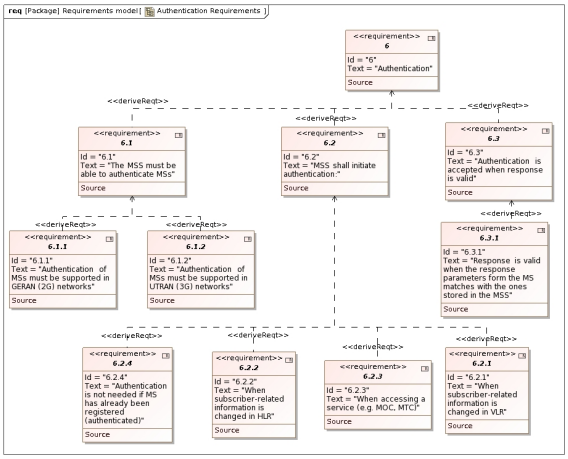


Figure 2: Example of a SysML requirements diagram

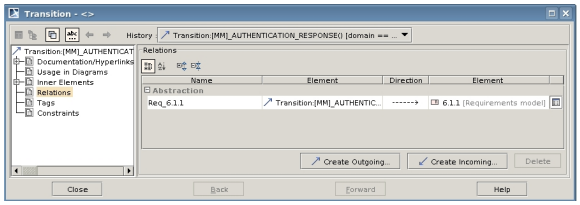


Figure 3: Linking requirements to a transition in a state machine

3.1.2. Requirements traced to models. The UML models of the SUT are built starting from the requirements models. During this process, the requirements are traced to different parts of the models to point how each requirement is addressed by the models. The relationships between requirements and models are specified on several levels. Non-leaf requirements are refined (linked) to models, e.g. state machine models. An exceptional situation is in the case of the top-level functional requirements, which are linked to use cases in the use case model of the SUT. The leaf requirements in the requirements tree are then linked to other UML elements to which they apply, e.g. transitions in a state machine or classes in a class diagram. Figure 3 shows how a requirement can be linked to a model element, e.g. a transition in a state machine using the MagicDraw editor.

This is done to ensure the traceability of requirements within the system models and to test cases.

These links are useful for evaluating (using the previously discussed validation rules) whether all the requirements have been reflected in the models or by showing what elements from different diagrams specify a given requirement. When all requirements have been linked to model elements and the models have been validated, the UML models are transformed into input for the Qtronic tool via an automated transformation.

The transformation [14] [15] basically translates UML models to the Qtronic Modeling Language (QML), the language used by Qtronic for specifying the SUT. QML is a textual specification language with a Java-like syntax in which one can specify the input/output ports of the system and what data types (complex data type are supported) can be send and received on different ports. The behavior of the SUT can be described either in QML or using a simplified version of UML statecharts. In the latter case, QML can be used as an action language for the statechart.

Qtronic provides support for requirement coverage during test generation. Requirements are associated to state models, more precisely to the actions on transitions via the `requirement` statement. Basically, the requirements in Qtronic are tags that are used to trace if a specific transition in the state model has been covered by the generated test cases.

During the transformation from UML to QML, links between requirements and model elements are preserved. In the current status of our work, only requirements attached to state machine transitions are propagated to Qtronic. Requirement hierarchy is specified in QML with the "/" character. Figure 4 shows an example of a state machine that has been transformed from UML to QML. In this figure, one can see that `requirement 6.1.1` and `requirement 6.1.2` in the MagicDraw model are propagated to the same transitions in QML. As the rest of the system description in QML is not relevant for this paper we do not include it here. However a detailed example can be found in [15].

3.1.3. Tracing requirements to tests. In Qtronic, test cases are generated according to different coverage criteria, like requirements coverage, transition coverage, state coverage. The coverage criteria are selected manually using the Qtronic user interface. By combining one or more of the mentioned criteria, Qtronic tries to generate tests based on those criteria. If the requirements coverage criterium is enabled, one can choose to test different requirements individually, by checking or unchecking the corresponding requirement in a list. Qtronic will then generate test cases that cover

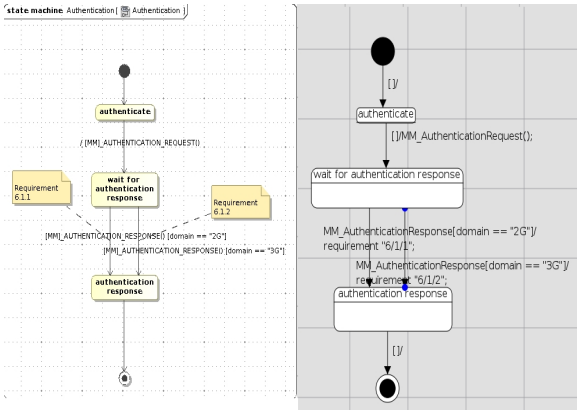


Figure 4: Example of a UML state machine in MagicDraw (left) and its equivalent in QML (right)

the selected requirements.

In the online testing mode, Qtronic handles the *test execution* process. One-by-one it generates an input message, sends it via the adapter to the SUT, and generates a new input message based on the responses from the SUT. A logging back-end can be used during test execution. The logging back-end provides connectivity to the Qtronic reporting infrastructure and it is used by Qtronic to generate a test report. Three logging back-ends are provided by default. With these logging back-ends, Qtronic can generate test reports in HTML, SQLite, and XML format. When all tests have been applied against the SUT, Qtronic generates a test report in the chosen format. Listing 1 shows an example of a generated test case specified in XML. As one can notice, the requirements have been propagated during test generation and included in the test specification (see line 6).

Listing 1: Requirement propagated to Qtronic test specification

```
1  <checkpoint>
2   <symbol value="transition: LocationUpdate-
        Authentication->LocationUpdate-Ciphering-
        initial-0-1"/>
3   <timestamp nanoseconds="447362000" seconds="0"/>
4  </checkpoint>
5  <checkpoint>
6   <symbol value="6 Authentication /1 The MSS must
        be able to authenticate MSs /1
        Authentication of MSs must be supported in
        GERAN (2G) networks"/>
7   <timestamp nanoseconds="447399000" seconds="0"/>
8  </checkpoint>
9  <checkpoint>
```

Figure 5 shows a example of a test report[1] generated

1. The test report also includes a requirements traceability matrix which we do not include due to space reasons.

Conformiq Qtronic Report

This HTML log has been automatically generated by Conformiq Qtronic[tm] from a system model.

The log contains both inputs to the system under test as well as the outputs from the system in addition t

Generated on Wed Apr 1 14:36:29 2009

Test Cases

Test Case	Test Verdict
Test Case Number 1 [MSC]	FAIL
Test Case Number 2 [MSC]	PASS

Qtronic Configuration [hide]

Qtronic Configuration	
System Model	/home/aton3/fabbors/Desktop/D-Mint/dmint-aau/case_st
Use Case Model	
Maximum Latency	2.83333
Model-driven Testing Heuristic	Coverage Directed
Automatic Pause	disabled
Single Test Run	disabled
Stop at 100% Coverage	disabled
State Coverage	enabled
Transition Coverage	enabled
2-Transition Coverage	disabled
Implicit Consumption	disabled
Boundary Value Analysis	disabled
Branch Coverage	disabled
Method Coverage	disabled
Statement Coverage	disabled
Parallel Transition Coverage	disabled
All Paths: States	disabled
All Paths: Transitions	disabled
All Paths: Control Flow	disabled

Coverage Information [hide]

Coverage	
Uncovered Requirements	'6 Authentication /1 The MSS must be able to authenticate MSs '6 Authentication /1 The MSS must be able to authenticate MSs '6 Authentication /1 The MSS must be able to authenticate MSs '7 Ciphering /1 The MSS must be able to cipher the communicat '7 Ciphering /2 Ciphering procedure is initiated by the MSS after

Figure 5: Test report produced by Qtronic

by Qtronic with a HTML logging back-end. It is also possible to inspect each test case individually by clicking on the *[MSC]* link next to the test case number in the test report. This will bring up a message sequence chart (MSC) showing the order of messages sent and received by Qtronic.

3.2. Back-tracing of requirements

The approach opposite to the one presented above, is to trace-back requirements from test cases to models. For this purpose, we analyze the test report, collect the information of the failed test cases, and trace the requirements attached to those test cases, back to system models. This way we can see which requirements were not validated during testing and to what parts of the specification they are linked.

We have developed a Python script that automatically analyzes the Qtronic test report and generates a set of OCL queries (see Figure 6), that we use in MagicDraw to locate erroneous parts in the UML system models. In this way, we can see which requirements failed during testing and to what model elements they are linked. Figure 7 shows how requirements, which

OCL Constraints

The following OCL 2.0 constraints where generated!

This expression is to trace uncovered or failed requirements in SysML requirements diagram:

not(Id = '3') and not(Id = '6') and not(Id = '6.1') and not(Id = '6.1.1') and not(Id = '6.1.2') and not(Id = '7') and not(Id = '7.1') and not(Id = '7.2')

This expression is to find uncovered or failed requirements placed on transitions:

not(clientDependency->exists(a | (a.supplier.name->includes('3') or a.supplier.name->includes('6') or a.supplier.name->includes('6.1') or a.supplier.name->includes('6.1.1') or a.supplier.name->includes('6.1.2') or a.supplier.name->includes('7') or a.supplier.name->includes('7.1') or a.supplier.name->includes('7.2'))) and clientDependency->notEmpty())

Figure 6: OCL constraints produced by the Req2Ocl script

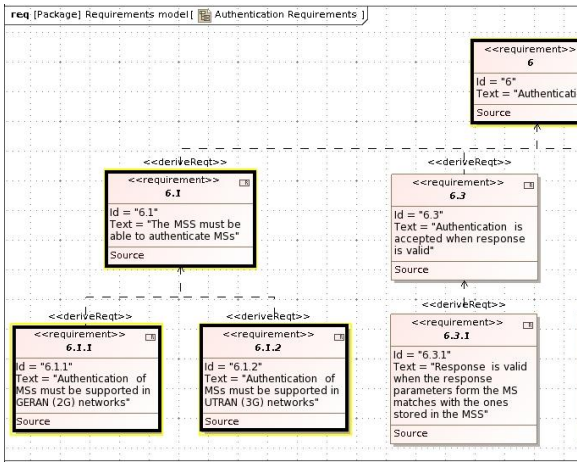


Figure 7: Tracing of requirements to a SysML Requirement Diagram

failed during testing, are found in the requirements model with the help of the OCL queries. In Figure 8 one can see how the same requirements are found in the state machine diagram, on the same transitions to which they were initially traced. Ultimately, since requirements are traced to model elements, it facilitates the identification of which functionalities of SUT are not in sync with the *model*, and hence with the requirements.

4. Conclusion

This paper has presented an approach for traceability of requirements in a model-based testing approach. We have shown how requirements can be traced to models, to test specifications, and back to models again. Traceability of requirements facilitates the process of locating parts in the system models that are causing failed test cases. Further, traceability of requirements can help in depicting missing tests, i.e. when critical requirements are not traced to any tests.

Currently, our presented approach only supports

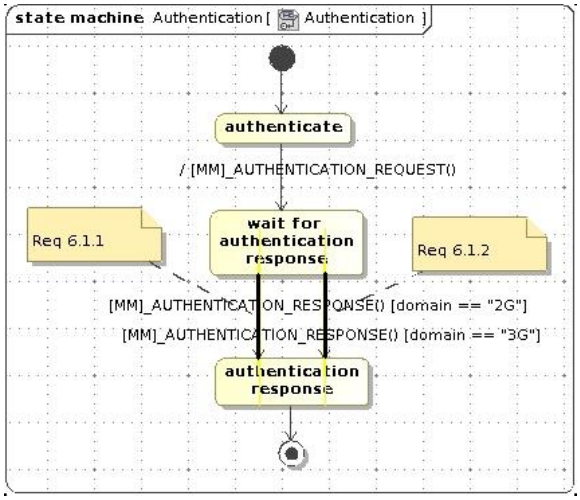


Figure 8: Tracing of requirements to transitions in a state machine

online testing. In the future, we will extend it to offline testing, as well. Another future goal is to find a solution for tracing non-functional requirements to system models and, respectively, to tests. This is something that has not yet been fully investigated.

Our approach benefits from a well integrated tool chain, in which specialized tools are used for each phase of the model-based testing process. When not already provided by the tools involved, we have provided automation of the transitions between the phases of the process, allowing to have a fast feed-back loop for testing and debugging the specifications or the implementation of the SUT. In addition, having a fully automated approach, the effort in updating the models and performing the testing again was diminished.

The approach also proved beneficial through the fact that many errors have been detected in the early stages of the process, when the system models have been created. The errors were caused mainly by omissions in the models and by misinterpreting the requirements. Thus, when failed test cases were reported after test runs we could focus our attention directly on debugging the implementation of the SUT.

References

[1] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause, "Rule-Based Generation of Requirements Traceability Relations," *The Journal of Systems & Software*, vol. 72, no. 2, pp. 105–127, 2004.

[2] C. Duan and J. Cleland-Huang, "Visualization and Analysis in Automated Trace Retrieval," in *Requirements Engineering Visualization, 2006. REV'06. First International Workshop on*, 2006, pp. 5–5.

[3] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability," in *13th IEEE International Conference on Requirements Engineering, 2005. Proceedings*, pp. 135–144.

[4] J. Hayes, A. Dekhtyar, and J. Osborne, "Improving Requirements Tracing via Information Retrieval," in *11th IEEE International Requirements Engineering Conference, 2003. Proceedings*, 2003, pp. 138–147.

[5] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting, "Requirements Traceability in Automated Test Generation: Application to Smart Card Software Validation," in *Proceedings of the 1st international workshop on Advances in model-based testing*. ACM New York, NY, USA, 2005, pp. 1–7.

[6] E. Bernard and B. Legeard, "Requirements Traceability in the Model-Based Testing Process," in *Software Engineering*, ser. Lecture Notes in Informatics, vol. 106. Bttinger, Stefan and Theuvsen, Ludwig and Rank, Susanne and Morgenstern, Marlies, 2007, pp. 45–54.

[7] Object Management Group, "OMG SysML Specification," Tech. Rep. [Online]. Available: http://www.omg.org/spec/SysML/1.1/

[8] "Unified Modeling Language - http://www.omg.org/spec/UML/2.0/." [Online]. Available: http://www.omg.org/spec/UML/2.0/

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

[10] "NoMagic MagicDraw," http://www.magicdraw.com/.

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

[12] T. Tsumaki and Y. Morisawa, "A Framework of Requirements Tracing using UML," in *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, 2000, pp. 206–213.

[13] R. Settimi, J. Cleland-Huang, O. Khadra, J. Mody, W. Lukasik, and C. DePalma, "Supporting software evolution through dynamically retrieving traces to UML artifacts."

[14] T. Pääjärvi, "Generation Input for the Test Generator Tool from UML Design Models," Master's thesis, Åbo Akademi University, 2009.

[15] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truşcan, and J. Lilius, "A Semantic Transformation from UML Models to Input for the Qtronic Test Design Tool," Turku Centre for Computer Science (TUCS), Tech. Rep. 942, 2009.

Paper II

Including Model-Based Statistical Testing in the MATERA Approach

Andreas Bäcklund, Fredrik Abbors, and Dragos Truscan

Originally published *2010 Proceeding of 3rd Workshop on Model-based testing in practice.*. IEEE. June 2010. Paris, France.

Including Model-Based Statistical Testing in the MATERA Approach

Andreas Bäcklund, Fredrik Abbors, and Dragos Truscan

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

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

1 Introduction

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

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

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

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

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

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

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

2 Related Work

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

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

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

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

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

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

3 Overview of MATERA

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

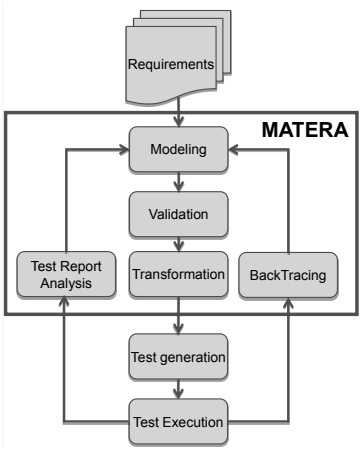


Fig. 1. MATERA process

4 Statistical Approach for MATERA

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

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

4.1 Requirements Modeling

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

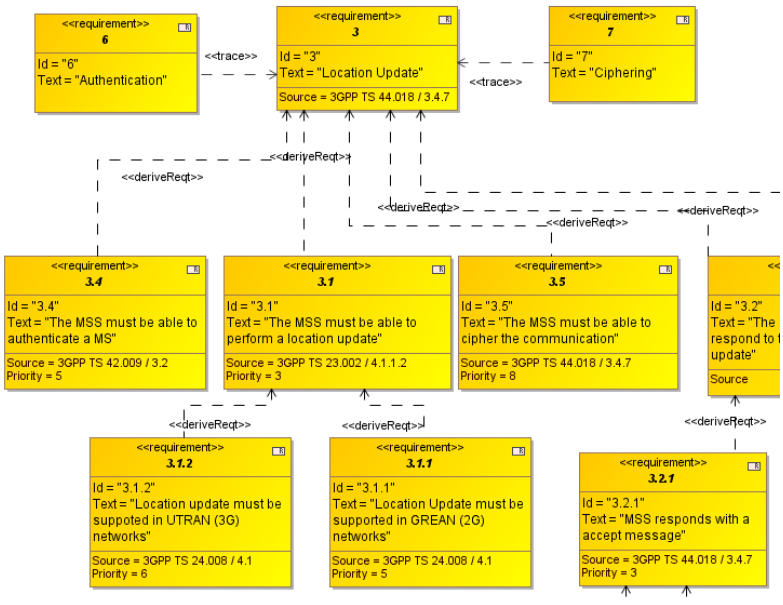


Fig. 2. Requirement Diagram with priorities

4.2 System Modeling

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

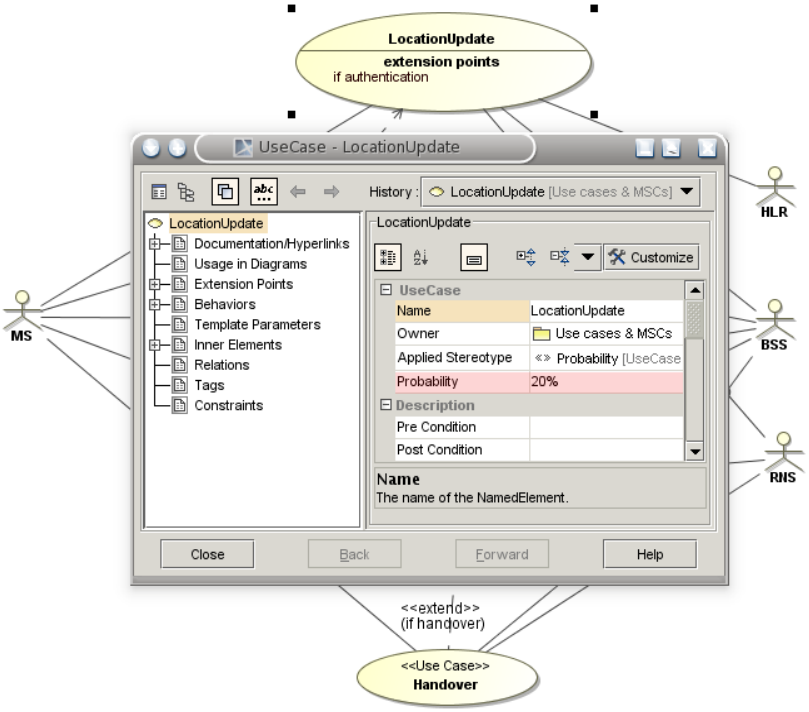


Fig. 3. Use case diagram with probability

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

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

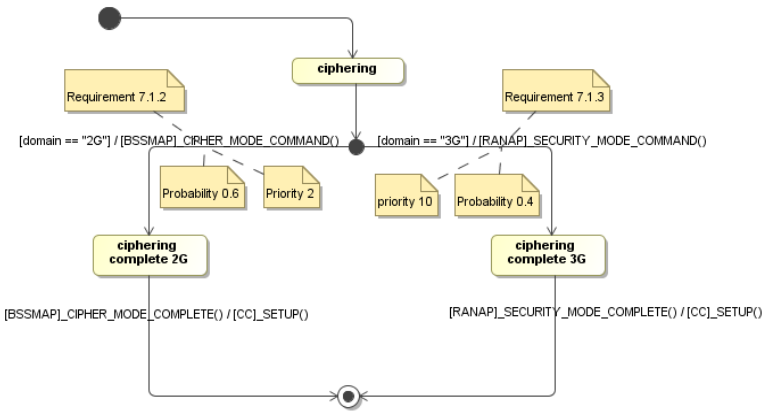


Fig. 4. UML state machine diagram

4.3 Test Case Generation

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

4.4 Test Case Ordering

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

4.5 Test Execution

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

4.6 Test Log Analysis

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

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

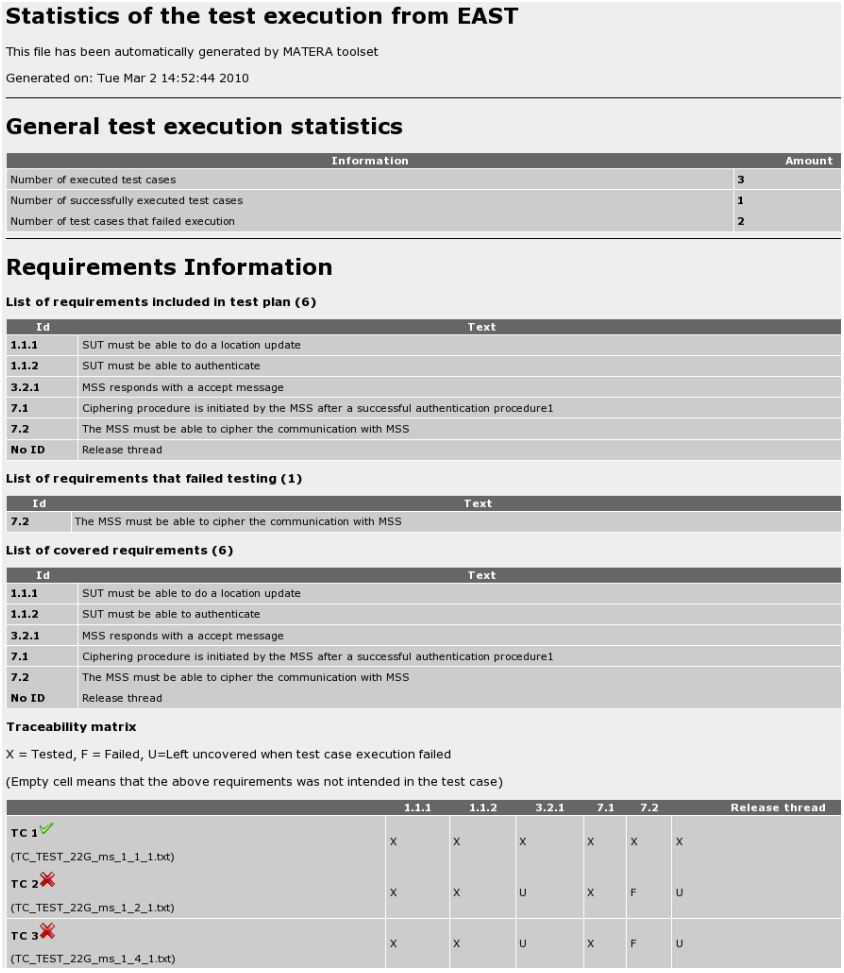
4.7 Feedback Loop

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

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

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

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



Statistics of the test execution from EAST

This file has been automatically generated by MATERA toolset

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

General test execution statistics

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

Requirements Information

List of requirements included in test plan (6)

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

List of requirements that failed testing (1)

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

List of covered requirements (6)

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

Traceability matrix

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

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

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

Fig. 5. Statistical Report

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

5 Tool Support

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

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

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

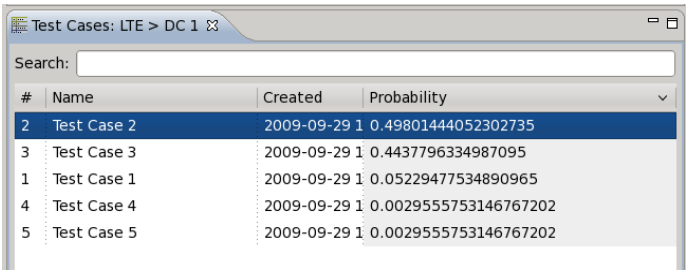


Fig. 6. Test case sequence ordered by weighted probability in Qtronic

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

6 Conclusions

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

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

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

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

References

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

Paper III

MATERA - An Integrated Framework for Model-based Testing

Fredrik Abbors, Andreas Bäcklund, and Dragos Truscan

Originally published *2010 Proceeding of 7th Workshop on System Testing and Validation*. IEEE. March 2010. Oxford, UK.

MATERA - An Integrated Framework for Model-Based Testing

Fredrik Abbors, Andreas Bäcklund, and Dragoş Truşcan
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520, Turku, Finland
Email: {Fredrik.Abbors, Andreas.C.Backlund, Dragos.Truscan}@abo.fi

***Abstract*—This paper presents MATERA, a framework that integrates modeling in the Unified Modeling Language (UML), with requirement traceability across a model-based testing (MBT) process. The Graphical User Interface (GUI) of MATERA is implemented as a plug-in in the NoMagic's MagicDraw modeling tool, combining existing capabilities of MagicDraw with custom ones. MATERA supports graphical specification of the requirements using SysML and tracing of them to the UML models specifying the SUT. Model validation is performed in MagicDraw using both predefined and custom validation rules. The resulting models are automatically transformed into input for the Conformiq Qtronic tool, used for automated test generation. Upon executing the test scripts generated by Qtronic in the NetHawk's East execution environment, the results of statistic analysis of the test run are presented in the GUI. The back-traceability of the covered requirements from test to models is also provided in the GUI to facilitate the identification of the source of possible errors in the models. The approach we present shows that existing model-based languages and tools are an enabler for model-based testing and for providing integrated tool support across the MBT process.**

***Keywords*-Model-Based Testing; Model Validation; Requirements Traceability;**

I. Introduction

The software industry is facing several challenges in delivering increased business value to their customers. Customers demand more flexible, reliable, low cost, and highly efficient software solutions. Additionally, the customers usually require the software systems to be deployed within different types of distributed environments. Research has shown that as much as 60 per cent of the total development time can be spent in testing [1]. This implies that testing is a highly expensive and time consuming process.

Model-based testing is a technique that tries to address these issues by introducing automatic generation of tests from models representing the behavior of the System Under Test (SUT). Using models for test generation increases the pressure put on the modeling process and on the quality of the models used for test generation, as any inconsistency in the models will reflect later on in the quality of the generated test cases.

In order to address these issues we suggested a modeling approach [2] which puts emphasis on three aspects. First, the models of the SUT are built in a systematic manner starting from requirements, after which they are fed as input to the test generation tools. Secondly, several model types are used to model different perspectives of the system like behavior, data, architecture, test configuration. Thirdly, the requirements of the SUT are traced through all the stages of the testing process and back-traced from the executed test cases back to models. The approach and the afferent tool support is also referred to as MATERA (Modeling for Automated TEst deRivation at bo Akademi).

In this paper, we describe how the tool support for the MATERA approach is provided by reusing and adapting existing commercial tools. More specifically, we try to show that existing model-based techniques and tool can become an enabler for supporting model-based testing and for providing integration across the MBT tool chain. MATERA has been targeted to the telecommunications domain and thus the examples used in this paper are excerpts from and industrial case study. Further information on case study can be found in [3].

II. MATERA

The MATERA tool-set is used to provide support for the approach by integrating modeling in the Unified Modeling Language (UML) [4] and requirement traceability across a custom MBT process (see Figure 1). As mentioned in the introduction, a set of models are created from the system requirements. These models are validated by checking that they are consistent and that all the information required by the modeling process is included. Consequently, the models are transformed into input for the test derivation tool. The resulting test cases are executed (after being implemented) using a test execution framework. The results of the test execution are analyzed and a report is generated. Requirements are linked to artifacts at different levels of the testing process and finally attached to generated test cases. This allows one to traceback to models which test cases have covered different modeling artifacts or from which part of the models a failed test case has originated.

In the following, we briefly present different features of the MATERA tool set accompanied by excerpts from a telecom case study.

A. Graphical User Interface

The GUI of the MATERA tool-set has been implemented as a plug-in in the MagicDraw UML modeling tool [5]. The plug-in is developed in Python using the Open Application

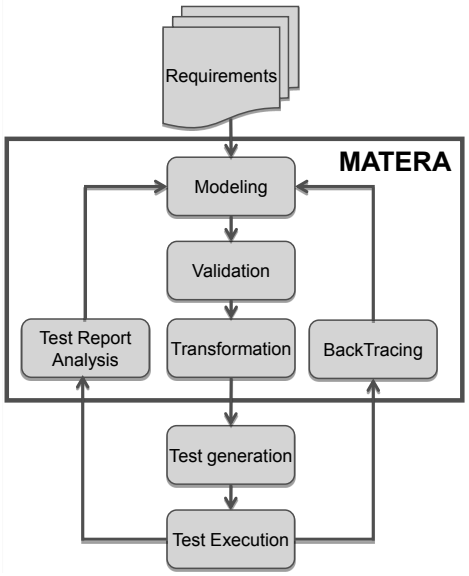


Figure 1. MATERA process

Programming Interface (Open API) of MagicDraw. The purpose of MATERA tool-set is to extend the capabilities of MagicDraw for specifying system models and using them as input for automatic test generation. Once the models are completely specified, they can be transformed to input for test generation tools. Besides model transformation, MATERA also supports model validation, test reporting, and (back-)tracing of requirements. Hence, MATERA promotes the integration of UML modeling with test generation tools. Figure 2 shows a caption of the MATERA GUI in MagicDraw.

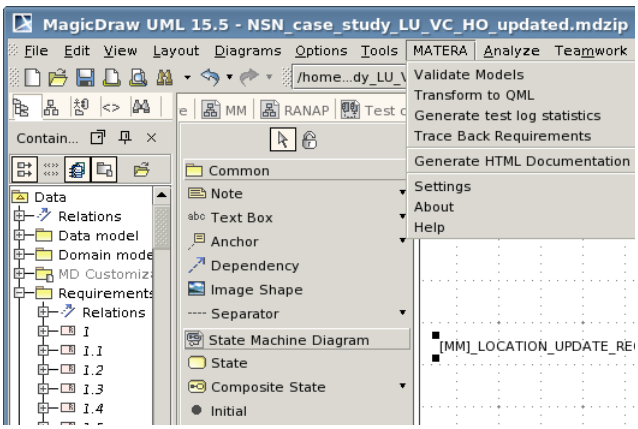


Figure 2. Caption of the MATERA GUI in MagicDraw.

B. Requirements Modeling

Requirements play an important role in any software project and it is also the starting point of the testing process [6]. MATERA starts with the analysis and structuring of the informal requirements into a Requirements Model. The Requirements Diagrams of the Systems Modeling Language (SysML) [7] are used for this purpose. We use MagicDraw's model editor to create, edit, and structure requirement elements. Requirements are organized hierarchically in a tree-like structure, starting from top-level abstract requirements down to concrete testable requirements. Figure 3 shows how requirements are structured in MagicDraw editor. Each requirement is described using a *Name* and an *Id*, a *Text* field explains the requirement, and a *Source* field points to the document or standard from which the requirement was extracted. Further, requirements can be related to each other using the relationships defined by SysML. For instance, requirements can be derived into other requirements using the *deriveReqt* relationship or related horizontally using the *trace* relationship.

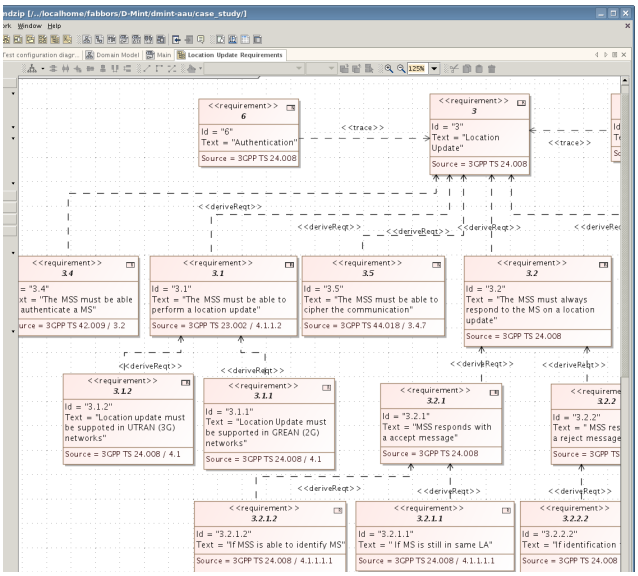


Figure 3. Structuring requirements in MagicDraw.

Traceability of requirements is a pivotal aspect of MBT that allows one to ensure that all requirements have been tested [8]. As the models are derived from requirements, it is important to track how different requirements are reflected in the models, on different perspectives, and on different abstraction levels. In MATERA, the requirements can be linked to different parts of the UML-based system specification, for instance to models or to model elements, to ensure requirements traceability throughout the process. When the specification is used for test generation, the requirements are associated with the generated test cases and propagated throughout test execution.

With MATERA it is possible to check that all specified requirements have been properly linked to models or model elements. For this purpose we use MagicDraw's validation

engine and custom Object Constraint Language (OCL) [9] rules to check e.g. that the leaf requirements are not left unlinked or that every requirement has a unique Id.

C. Modeling the SUT

In MATERA, we take advantage of the expressiveness and graphical capabilities of UML for creating the specification of the SUT. In our case the test model is derived from high-level development models, such that partial reuse of the development specification is enabled. In addition, the SUT is specified from several perspectives to enable a successful test derivation process. These perspectives of the SUT are modeled using the UML diagram editors provided by MagicDraw; a class diagram is used to specify an architectural model describing the static structure of system. The architectural model shows what domain components exist and how they are interrelated through interfaces. A behavioral model describes the dynamic behavior of the SUT using state machines. Data models are represented as class diagrams and are used to describe the data exchanged between different domain entities. Last but not least, test configuration models, represented as object diagrams, are used to describe specific test configurations and to set up initial values for the test components.

In MATERA, different parts of the specifications are linked together in order to specify dependencies. We use MagicDraw's property editor to link model elements and data together, for example, every messages specified on an interface in the domain model is linked to the corresponded class in the data model describing the structure of the message. Also the properties of the elements can be edited using the Specification editor, see Figure 4.

D. Model Validation

Humans tend to make mistakes and forget things. Therefore, to gain efficiency of using a MBT process and reducing the costs by discovering faults at an early stage, it is necessary to validate the models before using them to e.g. automatically generate code or test cases [8]. Hence, a set of modeling guidelines and validation rules have been defined for ensuring the quality of the models. Modeling guidelines are used to specify how different models are created from requirements or from other models, what information they should contain, how this information is related to the information present in the other models, etc.

Validation rules have been defined and implemented for both Requirements Models and for System Models for checking different quality metrics of the resulting models before proceeding to the test derivation phase. These rules ensure that the models are syntactically correct, they are consistent with each other, and that they contain the information needed in the later phases of the testing process. In MATERA, validation is prerequisite before transforming the models.

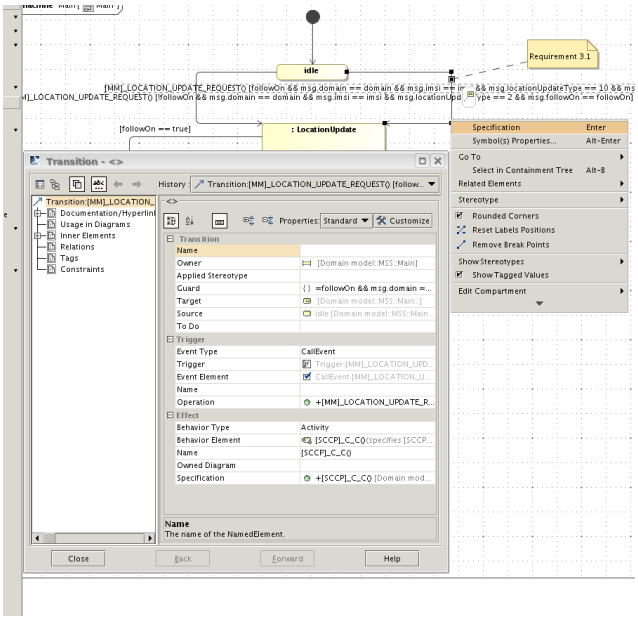


Figure 4. Screenshot showing the specification editor of a transition.

OCL is used to describe rules that apply to models. The rules describe conditions that must hold for the system being modeled. MagicDraw comes shipped with a set of predefined OCL rules for validating UML and SysML models. In addition to those, custom rules have been defined and implemented specifically for MATERA. The custom rules we created are all related to the modeling process, the application domain, and the specific MBT tool we target. For instance, we have created validation rules for checking the leaf requirements in the Requirements Diagram are linked to model elements and that messages defined on interfaces in the domain model are linked to data models.

An OCL rule is similar to a model element which has a number of editable properties e.g. name, specification, constrained element, severity level, etc. In order to provide reuse, rules are stored in different validation suites (packages) depending on the intended purpose of the rule, see Figure 6.

MagicDraw has a built-in validation engine for checking the rules against models. The validation in MagicDraw can be invoked at any time. When the validation is started the user will be prompted for the validation suite that he/she wants to apply, the scope (which models), and the severity level. Upon running the selected validation suite in the validation engine, MagicDraw creates a summary of the validation process as depicted in Figure 5, listing which elements are violating a rule and why. From this window the user can e.g. choose to open all diagrams with the elements violating a rule and see the faulty elements in the diagrams as they are highlighted. Once an error has been corrected the user can run the same validation suite again to see if the

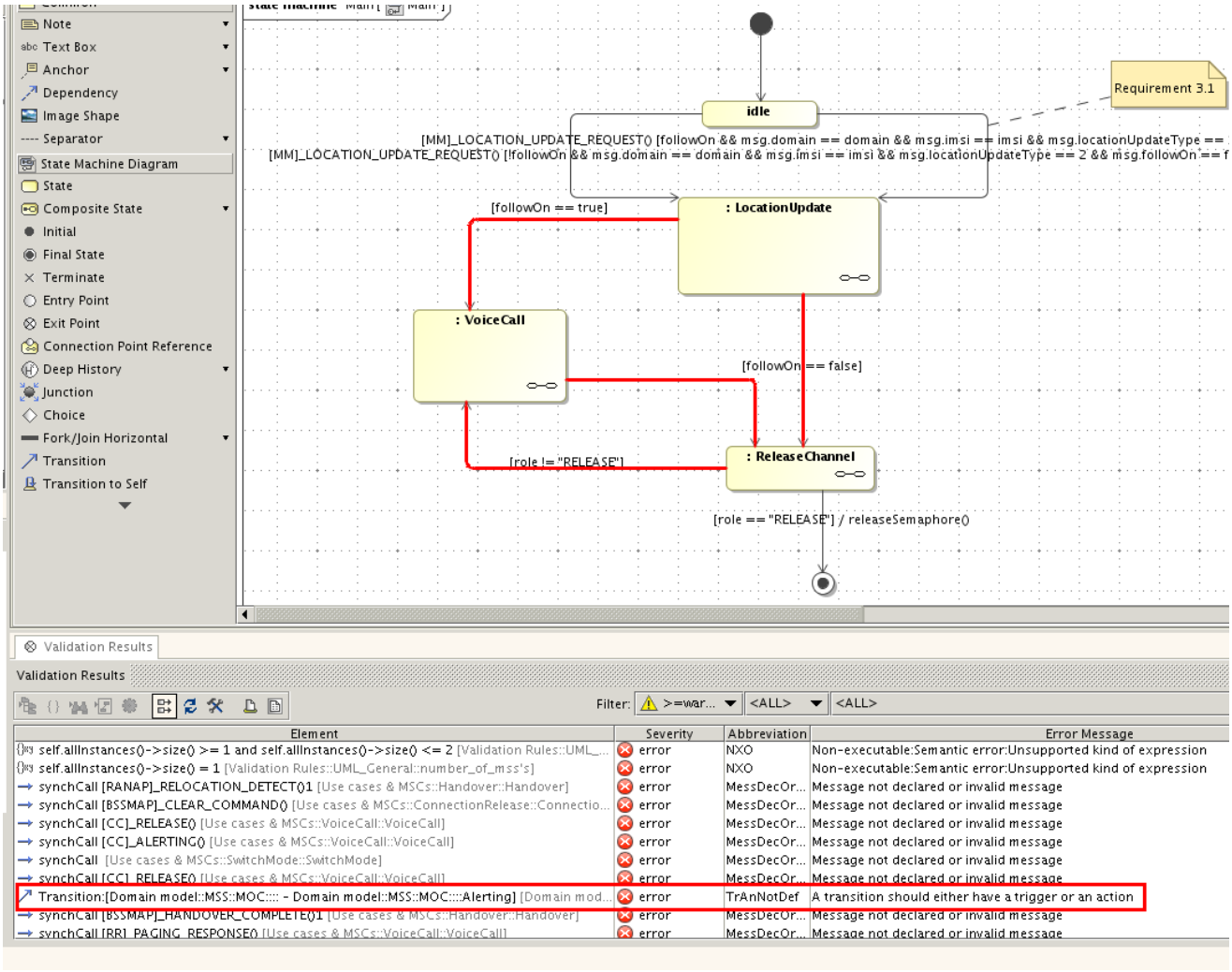


Figure 5. Validation summary.

modifications made any difference.

E. Transformation

In MATERA, the system models of SUT are transformed into input for the automated test derivation process. The transformation has two steps. First, the needed information from the models is collected by a parser module and stored into an internal representation. Then this information is read, by various build modules, and written (rendered) in the format supported by the test generation tool. The idea is to have a generic transformation approach and to be able to expand the approach to target different test generation tools. However, in our research we currently target only one particular test generation tool, namely Conformiq's Qtronic [10]. The transformation [11] [12] translates UML models to the Qtronic Modeling Language (QML), the language used by Qtronic for specifying the SUT.

The transformation also propagates requirement from UML models to QML. In QML, requirements are treated as textual tags attached to different parts of the specification, which are treating as testing goals during the test generations process. Once the test cases are generated they are implemented in the language used by the test execution tool (NetHawk's EAST [13] in our case) using the a scripting backend. During this process the requirements are propagated further and attached to executable tests allowing the test execution tool to trace and log the execution of tests cases and their associated requirements.

F. Test Reporting

MATERA offers support for test reporting. The test report will summarize the result of the testing process in terms of generated test cases, verdicts, coverage levels, etc. The information in the test report is collected form test logs and

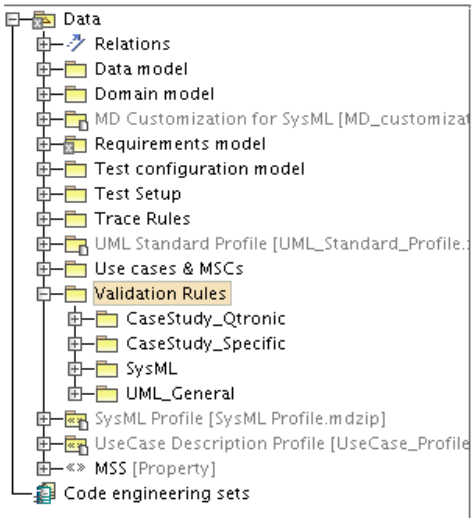


Figure 6. Validation suites in MATERA.

from the test scripts by comparing the test purposes encoded in the scripts against the results of the test execution.

When invoking the test report function from the MATERA menu (see Figure 2), a parser module collects and stores data from the test logs and scripts, similarly to the transformation. The collected data is then analyzed and presented to the user in HTML format, using the systems default HTML reader. The user only has to specify is the paths to the executed test scripts and their corresponding test logs. Figure 7 shows a snapshot of a test report.

G. Back-Tracing of Requirements

In MATERA, information from test logs is collected on how different requirements have been covered during both test generation and test execution phase, respectively. Based on this information the requirements are tracked back to the specifications from which the corresponding test cases have been generated, in order to detect the source of possible faults in the specifications. Upon selecting the *Trace Back Requirements* in the MATERA's GUI a Python script that analyzes the test logs and generates OCL queries that we use in MagicDraw to locate erroneous parts in the UML system models. The OCL queries are used to trace requirements from tests to the requirements models and to the requirements placed on transitions in state machines. The Python scripts generates OCL queries based on the information in the test logs and writes them to MagicDraw in a validation suite called "Trace Rules". We use again MagicDraw's OCL interpreter to find the requirements in the UML models based on the produced OCL queries.

This way one can see which requirements failed during testing and to what model elements they are linked. It also enables one to identify which parts of the system model have been covered by the test set. The back-tracing function in MATERA will highlight model elements in the system models, to which a failed requirement was linked. In Figure 8, the list of requirements covered by failed test cases in presented at the bottom of the screen. By selecting an entry in the list the corresponding requirement is highlighted in the diagram editor. Ultimately, since requirements are linked to model elements, it facilitates the identification of those parts of the models that are not in sync with the SUT, see Figure 9.

III. Related Work

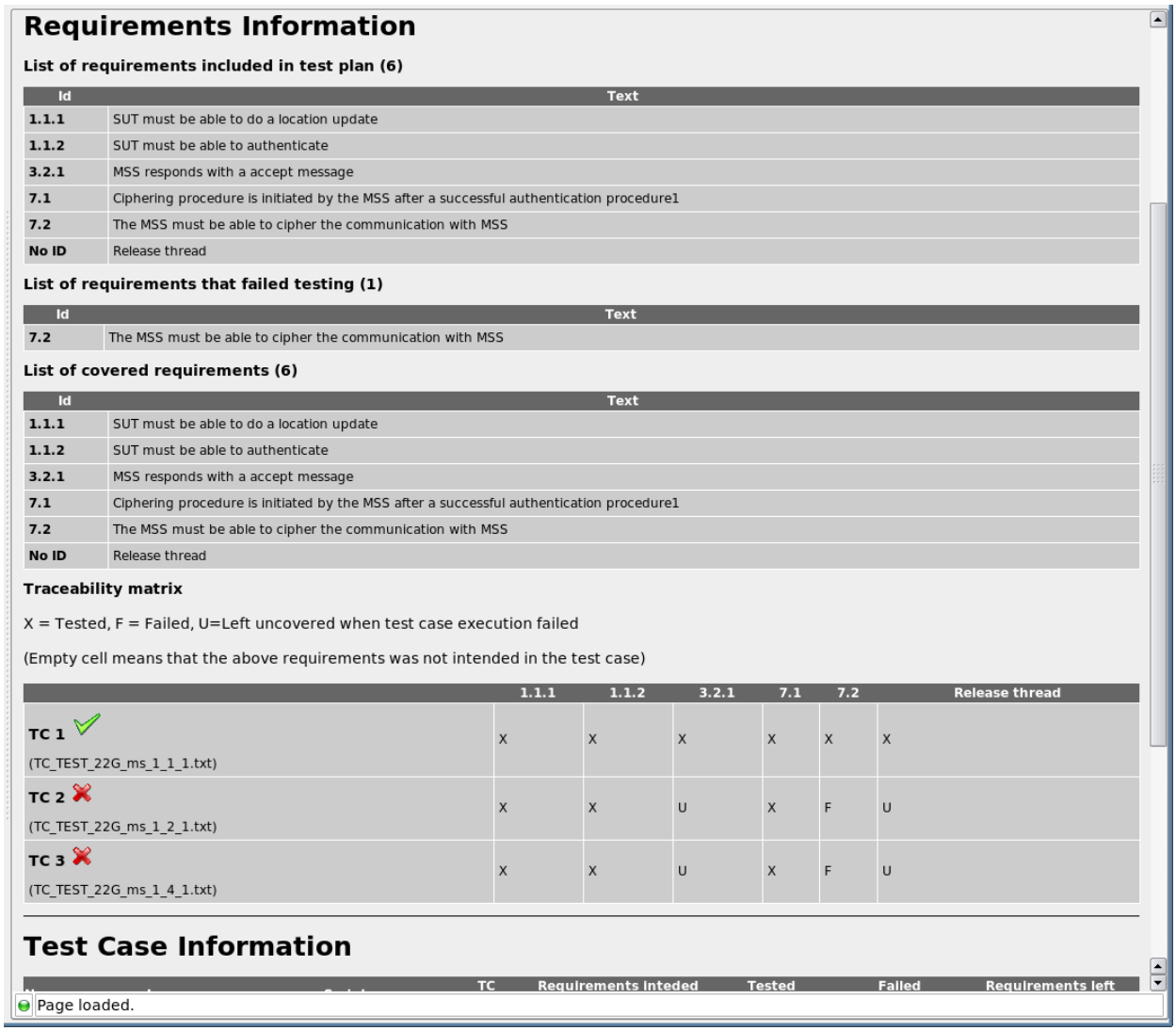
Similar research has been conducted within other industries. When modeling for automatic test generation, it is has proven beneficial to check the models against pre defined modeling rules and design guidelines before generating tests. In [14], the authors use the Object Constraint Language (OCL) to specify design guidelines and modeling rules for Simulink models. This approach is similar to ours, in the sense that rules written in the OCL language are used to check model consistency against a metamodel. However, their approach differs slightly from ours since the authors check the rules against a custom made Matlab/Simulink metamodel while we check OCL rules against the UML metamodel.

Other research similar to ours is described in [15]. In this research the authors use Matlab/Simulink behavioral models, instead of UML behavioral models, from where they automatically generate test sequences. Their approach differs somewhat from ours since it does not address automatic evaluation of test results.

From other reviewed works, the approach presented in [16] is closest to our approach. In there, the authors use a restricted set of UML diagrams together with OCL to describe both the static structure and dynamic behavior of the SUT. Requirements traceability is addressed by manually tagging the UML specification with ad-hoc comment symbols to associate a requirement with an OCL statement. Using the LEIRIOS (now Smartesting) Test Designer tool the authors automatically generate test cases out of the UML system specification and a traceability matrix is obtained after test execution. However, the Test Designer tool does not offer support for tracing requirements from test cases to the UML specification.

IV. Conclusion

This paper has presented a framework for integrating UML and requirements traceability in a MBT process. The MATERA framework is implemented as a plug-in for MagicDraw, which adds extended functionality to use UML models for automatic test generation. UML modeling is combined in MATERA with consistency checking of the specification using pre- and custom defined consistency rules, with the purpose of increasing the quality of the



Requirements Information

List of requirements included in test plan (6)

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

List of requirements that failed testing (1)

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

List of covered requirements (6)

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

Traceability matrix

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

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

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

Test Case Information

TC	Requirements inteded	Tested	Failed	Requirements left

Page loaded.

Figure 7. Caption of a test report generated from MATERA.

specifications used for automated test generation. Requirements are traced across the entire testing process; from models to test cases, and from test cases back to models. Requirements traceability is facilitated in MATERA by the back-traceability function.

The MATERA framework provides value by allowing testers to generate input for automatic test generation tools from a graphical representation of the SUT. The graphical representation is created using UML, which is one of the commonly used standard in the software industry. Additionally, back-tracing of requirements allows for having a visual overview of requirements that have not been properly tested. MATERA also provides the tester with a test report presented in HTML format which contains statistics of the test execution.

In our current research, we have focused mainly on generating input for Conformiq's Qtronic test generation tool. However, in the future we plan to target other test generation tools as well. Another future goal is to include statistical information into the UML models, based on past test executions, to be able to prioritize test cases or to focus the testing on specific parts of the system specification. We will also investigate how the information contained in the UML models can be used for generating adapter frameworks for different MBT tools.

ACKNOWLEDGMENT

Financial support from Tekes under the ITEA2 D-Mint project is gratefully acknowledged.

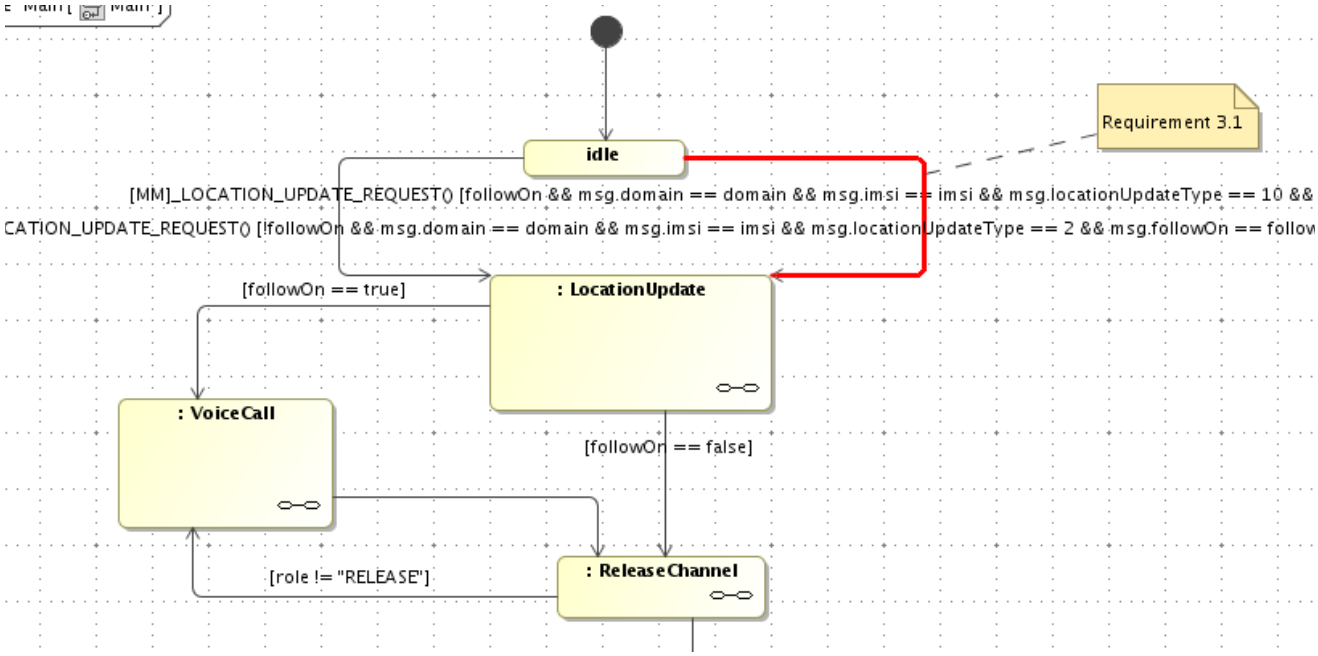


Figure 8. Back-tracing of a requirement to a transition.

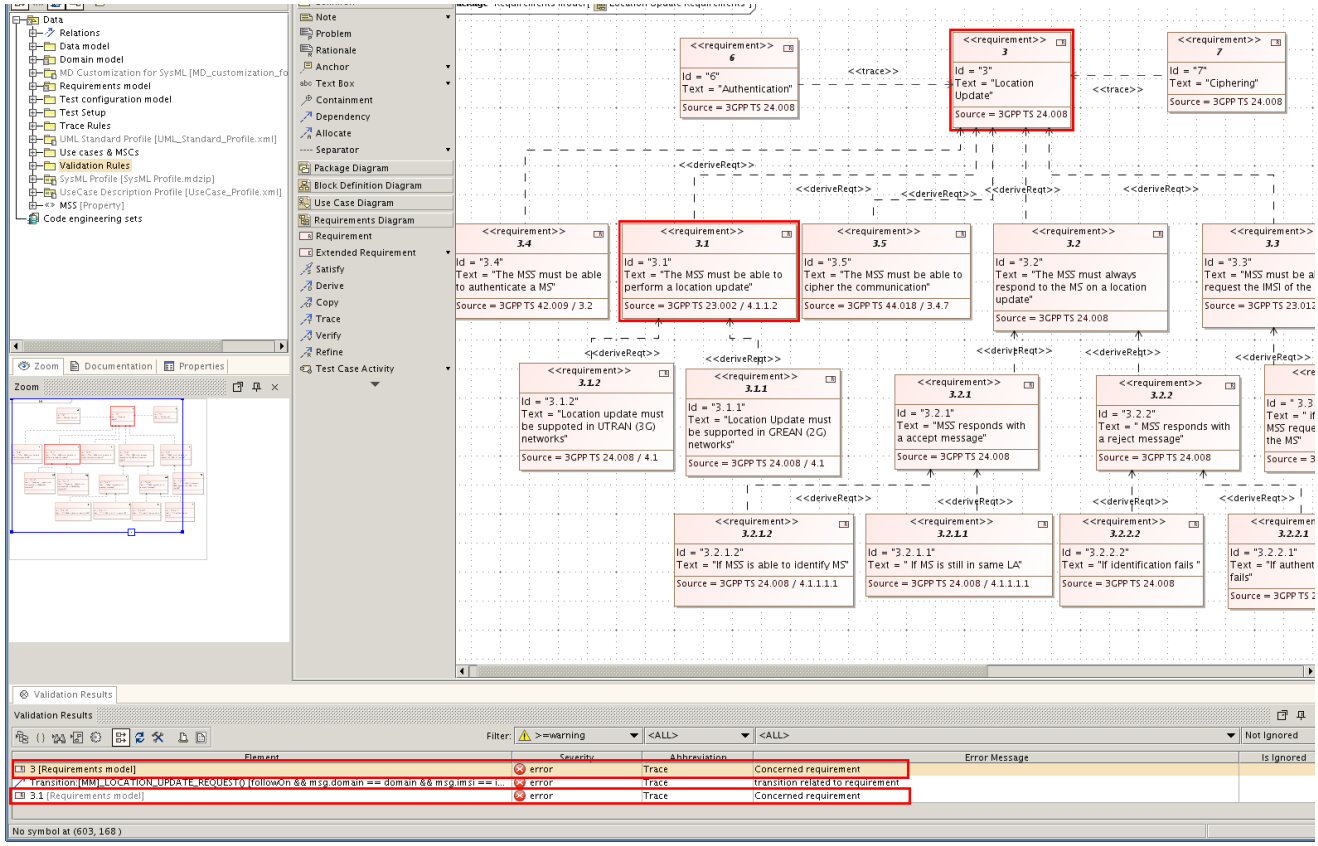


Figure 9. Back-tracing of requirements to the Requirement Diagram

.

REFERENCES

[1] "Historical Perspective in Optimising Software Testing Efforts - http://www.indianmba.com/Faculty_Column/FC139/fc139.html." [Online]. Available: http://www.indianmba.com/Faculty_Column/FC139/fc139.html

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

[3] F. Abbors, "An Approach for Tracing Functional Requirements in Model-Based Testing," Master's thesis, Åbo Akademi University, 2009.

[4] "Unified Modeling Language - http://www.omg.org/spec/UML/2.0/." [Online]. Available: http://www.omg.org/spec/UML/2.0/

[5] "NoMagic MagicDraw," http://www.magicdraw.com/.

[6] G. Fournier, *Essential Testing: A Use Case Driven Approach*. BookSurge Publishing, 2007, pp. 67–75.

[7] Object Management Group, "OMG SysML Specification," Tech. Rep. [Online]. Available: http://www.omg.org/spec/SysML/1.1/

[8] M. Utting, *The Role of Model-Based Testing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 510–517.

[9] *Object Constraint Language v2.0*, OMG, May 2006, http://www.omg.org/spec/OCL/2.0/PDF.

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

[11] T. Pääjärvi, "Generation Input for the Test Generator Tool from UML Design Models," Master's thesis, Åbo Akademi University, 2009.

[12] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truşcan, and J. Lilius, "A Semantic Transformation from UML Models to Input for the Qtronic Test Design Tool," Turku Centre for Computer Science (TUCS), Tech. Rep. 942, 2009.

[13] NetHawk, "NetHawk EAST," 2009. [Online]. Available: www.nethawk.fi/products/nethawk_simulators/nethawk_ims_tester/

[14] T. Farkas, C. Hein, and T. Ritter, "Automatic Evaluation of Modeling Rules and Design Guidelines," in *proc. of the Workshop "From code centric to Model centric Soft. Eng.", http://www.esi.es/modelware/c2m/papers.php*, 2006.

[15] M. Conrad, H. Dörr, I. Stürmer, and A. Schürr, "Graph Transformations for Model-based Testing," *GI-Lecture Notes in Informatics, P-12*, pp. 39–50, 2002.

[16] E. Bernard and B. Legeard, "Requirements Traceability in the Model-Based Testing Process," in *Software Engineering*, ser. Lecture Notes in Informatics, vol. 106. Bttinger, Stefan and Theuvsen, Ludwig and Rank, Susanne and Morgenstern, Marlies, 2007, pp. 45–54.

Paper IV

Applying Model-based Testing in the Telecommunications Domain

Fredrik Abbors, Veli-Matti Aho, Jani Koivulainen, Risto Teittinen, Dragos Truscan

Originally published *2012 Model-Based Testing for Embedded Systems.* Taylor and Francis Group, LLC. 2012.

to learn how to obtain a license.

17

Applying Model-Based Testing in the Telecommunication Domain

Fredrik Abbors, Veli-Matti Aho, Jani Koivulainen, Risto Teittinen, and Dragos Truscan

CONTENTS

It is in the public domain that model-based testing (MBT) has been used for years and its benefits have been emphasized by numerous publications.* Despite this, there is still one question pending. If MBT is an excellent way to test, why has MBT not made a major breakthrough in the telecommunication industry? In order to find the answer to this

*Results of case studies on industrial context have been explained in numerous publications. Dalal et al. described case studies and experiences in (Dala et al. 1999), and Prenninger, El-Ramly, and Horstmann listed and evaluated selected case studies in Prenninger, El-Ramly, and Horstmann (2005).

question, a MBT methodology was developed for a product testing project at Nokia Siemens Networks (Network 2009).

A systematic methodology was developed during the project. The methodology uses a *system model* instead of test models in contrast with many other MBT systems. In the context of the methodology, the term *system model* refers to models that define the behavior of the system under test (SUT) instead of the behavior of test cases.* In addition, during the development, the methodology was constantly evaluated to understand benefits and problematic aspects of the MBT technology.

The methodology includes a process and the supporting tool chain. The methodology was developed for testing functionality of an MSC Server (Mobile Services Switching Centre Server), a network element, using a *functional testing* approach, that is, the product was tested using a *black-box testing* technique.† The methodology exploited a so-called *offline testing* approach, where test cases were generated from the models before execution, instead of using an *online testing* approach, where models are interpreted step by step and each step is executed, instantly.‡

The content of this chapter will first provide an overview of the project in Section 17.1 that helps understand the following subsections within the chapter. Sections 17.2 through 17.4 focus on process, model development, validation, and transformation aspects. Sections 17.5 through 17.6 describe test generation and test execution aspects. Section 17.7 is devoted to requirement traceability as it spans across the entire process and the tool chain. Conclusions are provided in Section 17.9.

17.1 Overview

Firstly, the overview will provide a high-level view of the process and tools. Next, the SUT and its characteristics are explained. These together provide a detailed context for the work.

17.1.1 Process and tools

The high-level process used in the project is depicted in Figure 17.1. The top level of the process uses the MATERA approach (Abbors 2009a). Four major phases can be identified from the process. First, the requirements are processed and models describing the SUT are created. The models are validated using a set of validation rules in order to improve the quality of the models. Second, the tests are generated from the models. The test generation phase produces executable test scripts. Third, the test scripts are executed with the help of a test execution system. The execution phase produces test logs that are used for further analysis. Fourth, the tests are analyzed in case of failures and requirement coverage tracing is performed. The analysis exploits the test logs and the models. The phases of the process are described in detail in Sections 17.2 through 17.5. In addition, requirement traceability

*In case the models represent behavior of the tests, the term *test model* is often used. The differences in the two modeling approaches are explained in Chapter 2, Section 2.3 and by Malik et al. (2009).

†Beizer defines *functional testing* as an approach that considers implementation as a black box. Hence, the functional testing is often called black-box testing (Beizer 1990).

‡Utting, Pretschner, and Legeard describe offline testing in Utting, Pretschner, and Legeard (2006) as a method, where tests are strictly generated before the execution, in contrast with online testing, where a test system reacts on outputs of a SUT dynamically. In the context of model-based verification and MBT the terms *online* and *on the fly* are often used to refer to similar kinds of test execution approaches. Bérard et al. describe the on-the-fly method in Bérnard et al. (2001) as a way to construct parts of a reachability graph on a needs basis rather than constructing a full graph immediately.

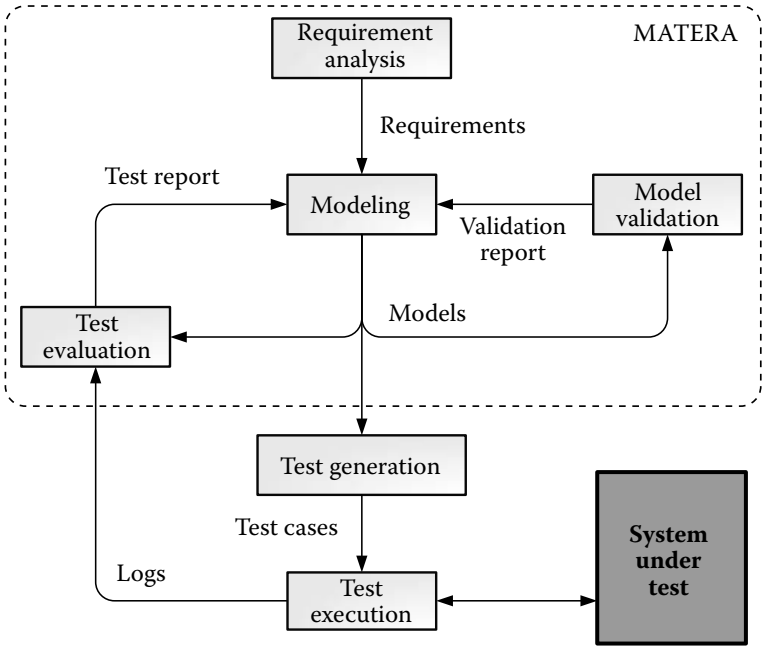


FIGURE 17.1
Process description overview.

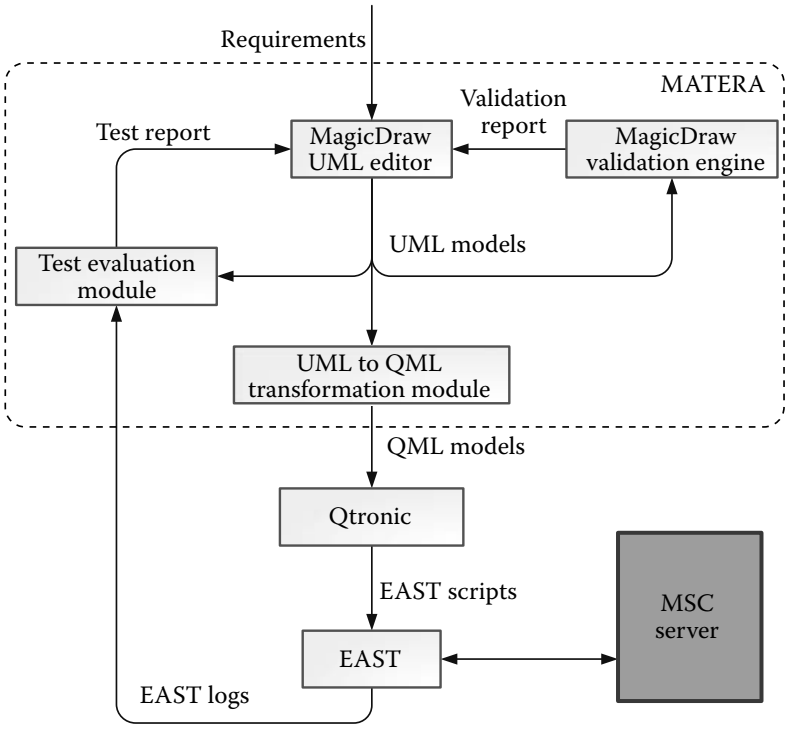


FIGURE 17.2
Tool chain overview.

is discussed in Section 17.6 as it is not restricted to any particular phase but spans across all the phases.

The process is supported by the tool chain depicted in Figure 17.2. The figure resembles Figure 17.1 and therefore indicates the roles of the tools with respect to the described process. The top level is supported by the MATERA framework (Abbors, Bäcklund, and

Truscan 2010), which is developed as a plug-in to the No Magic MagicDraw tool (Magic 2009). In MATERA, the Unified Modeling Language (UML) (Object Management Group models d) is edited, validated, and transformed to Qtronic Modeling Language (QML) (Conformiq 2009b) models and given as an input into the Conformiq Qtronic tool (Conformiq 2009a) for test generation. Qtronic outputs test scripts that are executed with Nethawk EAST (Nethawk 2009). The test logs produced by EAST are analyzed and evaluated against the original models using the MATERA test evaluation function. The relevant details of the tools are provided in Sections 17.2 through 17.5.

The UML models are edited with No Magic MagicDraw tool (Magic 2009). In addition, MagicDraw is used for model validation via custom rules implemented using the Object Constraint Language (OCL) (Object Management Group b). The UML models are transformed with a script into QML models and given as an input into the Conformiq Qtronic tool (Conformiq 2009a) for test generation. Qtronic outputs test scripts are executed with Nethawk EAST (Nethawk 2009). The test logs produced by EAST are analyzed and evaluated against the original models using a test evaluation script. The relevant details of the tools are provided in Sections 17.2 through 17.5.

17.1.2 System under test

The project focused on testing the Mobility Management (MM) feature of a MSC Server, that is, the MSC Server acted as the SUT. It is a key network element of second and third generation mobile telecommunication networks that establishes calls and controls handovers during calls. The MSC Server is capable of handling up to several million users and at the same time provides a near zero downtime. The MSC Server communicates with a number of other network elements as illustrated in Figure 17.3.

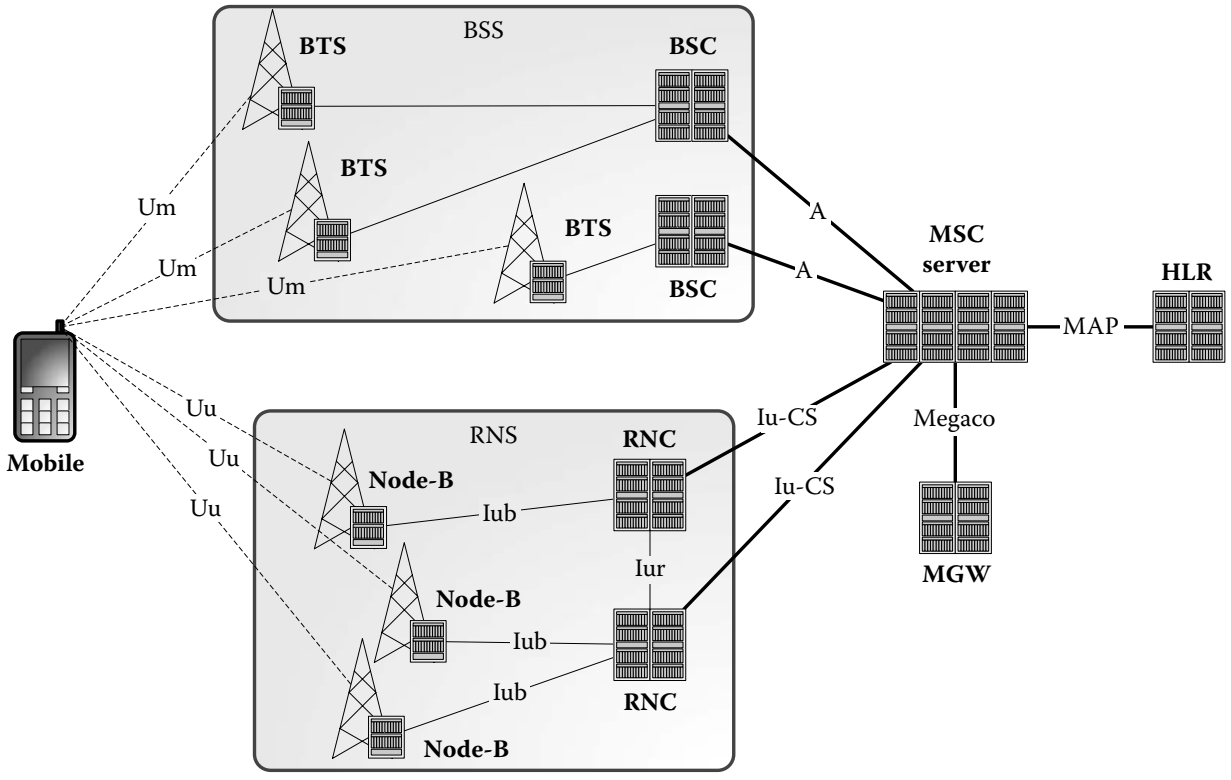


FIGURE 17.3
Project-related mobile telecommunication network elements.

The MSC Server is a typical telecommunication network element from a testing point of view. It has multiple interfaces with other network elements. In addition, the MSC Server communicates with multiple network elements of the same kind, for example, a MSC Server connects to many Radio Network Controllers (RNCs) and Base Station Controllers (BSCs). Communication between the network elements is concurrent and performance scalability aspects are already taken into account in the architecture of the network. The MSC Server also communicates with mobile phones using logical connections, that is, the MSC Server does not have a physical connection with the mobile phones, but the MSC Server uses services provided by other network elements. These elements are part of the radio access networks, that is, Base-Station Subsystem in the second generation network and Radio Network Subsystem (RNS) in the third generation network. The details of the network architecture are specified in the 3GPP Technical Specification 23.002 (The 3rd Generation Partnership Project 2005). Evolution from the second generation GSM systems to the third generation UMTS networks and a detailed description of the latter technology are provided by Kaaranen et al. (2005).

17.2 UML/SysML Modeling Process

The models of the SUT are created following the MATERA approach. The approach starts from the textual requirements of the system, the MSC Server, and incrementally builds a collection of models describing the SUT from different perspectives. In this context, textual requirements refer to the collection of stakeholder requirements, as well as additional documents such as protocol specifications, standards, etc. The modeling phase captures several perspectives (architecture, behavior, data, and test configuration) of the system, at several abstraction levels, by spanning the Functional View and Logical View layers described in Figure 17.4. The functional view defines how the system is expected to behave when it interacts with its users. The logical view describes the logical parts of the system, with behaviors and interactions, without relating to the actual implementation of the system. Each perspective is initially specified on the functional view via the feature, requirements, use case, and sequence diagrams, and it is subsequently refined on the logical view (state machine diagrams, class diagrams, and object diagrams). There are both horizontal (between the perspectives on the same level) and vertical relationships (refinements) among the specification artifacts in this process, as will be illustrated throughout this section.

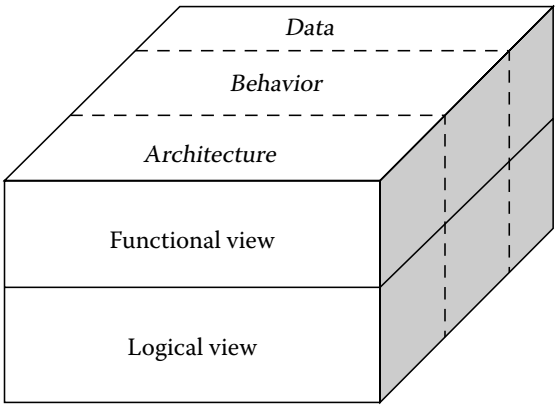


FIGURE 17.4
Modeling perspectives in the NSN case study.

The UML (Object Management Group d) is used as a specification language for system modeling in the project. Additionally, the *requirements diagrams* of the Systems Modeling Language (SysML) (Object Management Group c) are used to capture the requirements of the system in a graphical manner. The No Magic MagicDraw tool was employed to create the models. MagicDraw is a commercial software and system modeling tool, which offers support for both UML and SysML. The tool also offers support for automatic code generation, model validation, model analysis, reporting, etc., and can be extended with the use of various plug-ins.

The system models are created in a systematic manner, based on the MATERA guidelines (Figure 17.5), starting from the textual requirements. The approach consists of five phases. In each phase, a new set of models is created. Each model describes the system from a different perspective. The approach is iterative, so each phase can be visited several times, and the models are constructed incrementally.

The *first phase* deals with the identification of stakeholder requirements, standards, and associated specifications. Since the models are derived from requirements, it is necessary to identify and collect as much relevant information about the system as possible. The system models are later built based on the collected information.

In *phase two*, *feature models* and *requirements models* are created from the information collected in the previous phase. Initially, the features of the SUT are specified using UML Class diagrams (Figure 17.6). The feature models are mainly derived from product requirements and they give a rough outline of which features and functionality the system must be able to perform. Each class describes one feature, whereas *mandatory* and *optional* relationships between features are modeled using aggregation and composition relationships between classes. The feature diagram follows the principles of functional decomposition, where high-level features are decomposed into subfeatures.

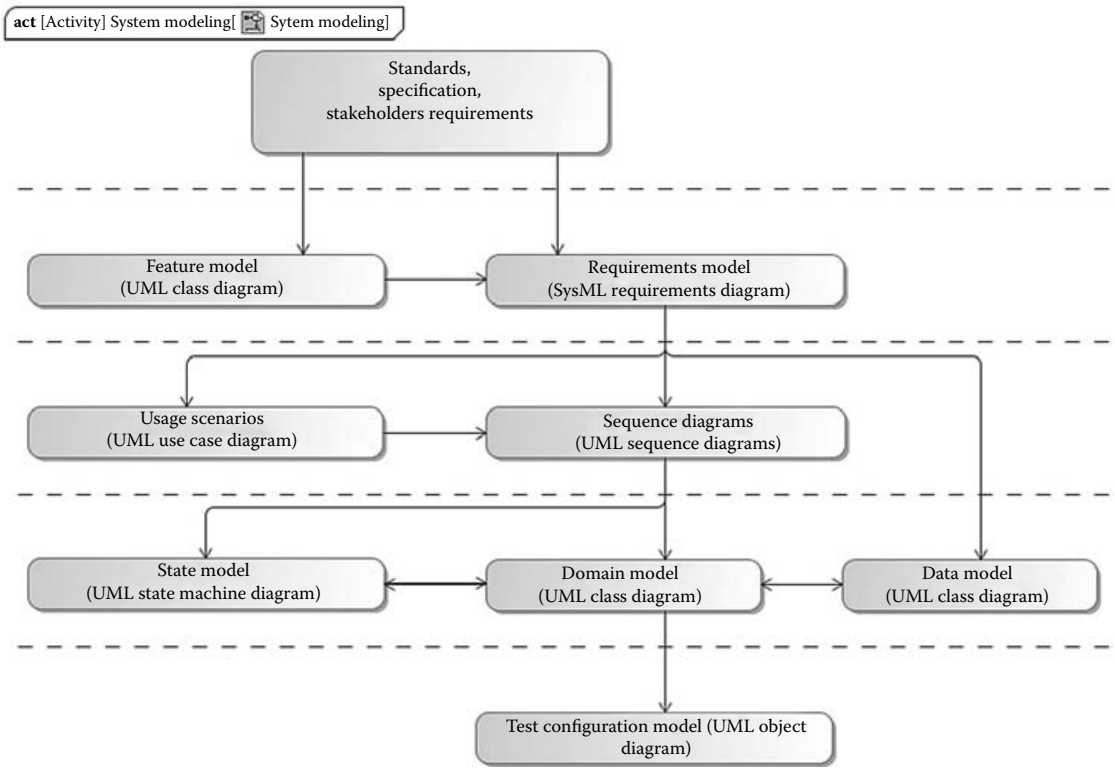


FIGURE 17.5
System modeling process.

A set of requirement models are created using SysML Requirements Diagrams (Figure 17.7). The requirements are specified on several levels of abstraction following principles of functional decomposition. One such model is created for every leaf in the feature diagram. The purpose of these models is to structure the *requirement specifications* corresponding to each feature in a graphical manner. They are structured in requirements diagrams similar to a UML class diagram in which the classes are annotated with the ≪`requirement`≫ stereotype. A requirement in SysML is specified using different properties including an *id* field, a *textual description*, and the *source* of the requirement. The *textual description* gives a brief explanation of the requirement, while additional details (e.g., technical specifications) are added in the documentation field of each requirement (not visible in the previous figure). The *source* field directs the document to where the requirement has been extracted from.

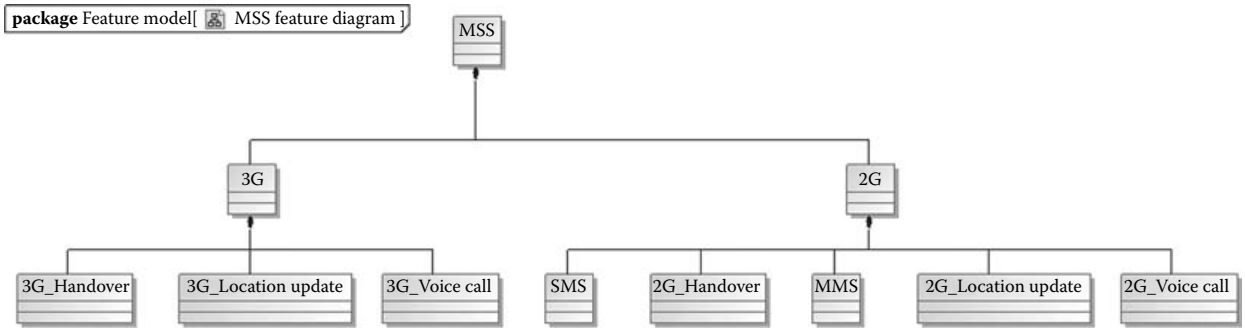


FIGURE 17.6
A feature diagram of the MSC Server.

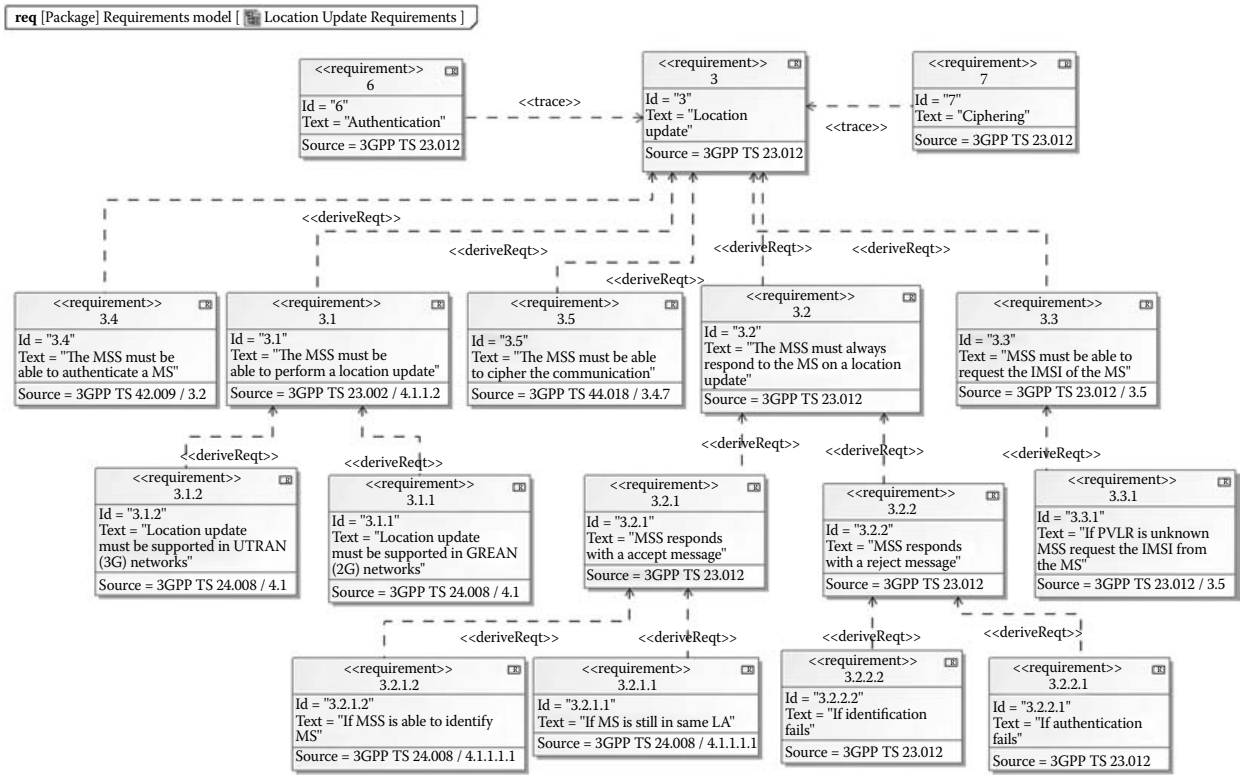


FIGURE 17.7
SysML requirements diagram. (Reproduced from Abbors, F., Backlund, A., and Truscan, D., MATERA—An Integrated Framework for Model-Based Testing, *Proceedings: 2010 17th IEEE Conference and Workshops*, © 2010 IEEE.)

Requirements are traced to other requirements on the same abstraction level or between requirements and other model elements in UML by using relationships such as *DeriveReqt*, *Satisfy*, *Verify*, *Refine*, *Trace*, or *Copy*. If necessary, requirements are also arranged into different categories such as functional, architectural, and communication (data).

In the *third phase*, a *use case model* and a set of *sequence diagrams* are created. The purpose of the use case model (Figure 17.8) is to present a graphical overview of the main functionality of the system. The use case model also identifies the border of the system and the external entities (actors) with which the system must interact. Each use case has a detailed textual description using a tabular format (see Figure 17.9). The description includes fields for precondition, postcondition, actors using the use case, possible sub-use cases, as well as an abstract textual description of the sequence of actions performed by the system when the scenario modeled by the use case is in use. Message sequence charts (MSCs), or sequence diagrams, are illustrated in Figure 17.10 and are primarily used to describe the interactions between different entities in a sequential order, as well as for describing the behavior of a use case, by showing the messages (and their parameters) that are passed between entities for a given use case. Basically the intended usage of MSC is twofold: to discover entities communicating with the system and to identify the message exchange between these entities. The messages exchanged between entities (referred to as *lifelines* in the context of sequence diagrams) are extracted from the protocol specifications referenced by the requirements.

In the *fourth phase*, we define the *domain*, *data*, and *state* models of the SUT. The domain model (Figure 17.11) is represented as a class diagram showing the domain entities as *classes* and their properties (*attributes*). The domain model also describes the *interfaces* that the domain entities use for communicating with one another. Each interface contains a set of messages that can be received by an entity, modeled as class operations. The names of the operations are prefixed with the acronym of the protocol level at which they are used, similar to the approach followed in the sequence diagrams. For instance, the

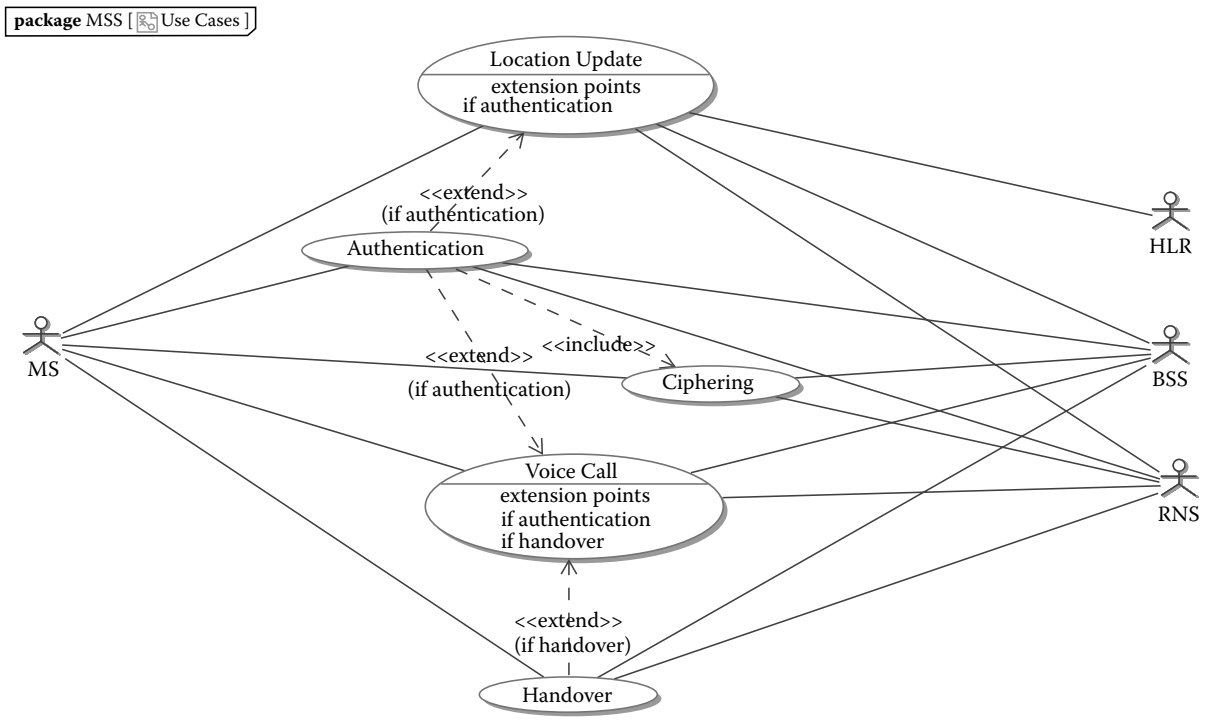


FIGURE 17.8
Use case diagram of the MSC server.

Name	Location Update		
Author	Fredrik Abbors		
Date	1.10.2008		
Actors	MS, HLR, BSS, RNS		
Sub-cases	Authentication, Ciphering		
Description	The MS requests a location update from MSS. The MSS can in some cases initiate authentication and ciphering of the MSS. Finally, the MSS will respond to MS with the location area.		
Pre-conditions	Connection established between MS and MSS.		
Post-conditions	The location area of MS stored/updated in MSS's registers.		
Scenario		Actor input	System response
	1	MS sends requests to MSS to update its location	
	2		MSS responds and accept message containing information about the location area
	3	MS responds with an acknowledge message	

FIGURE 17.9
Tabular description of the location update use case.

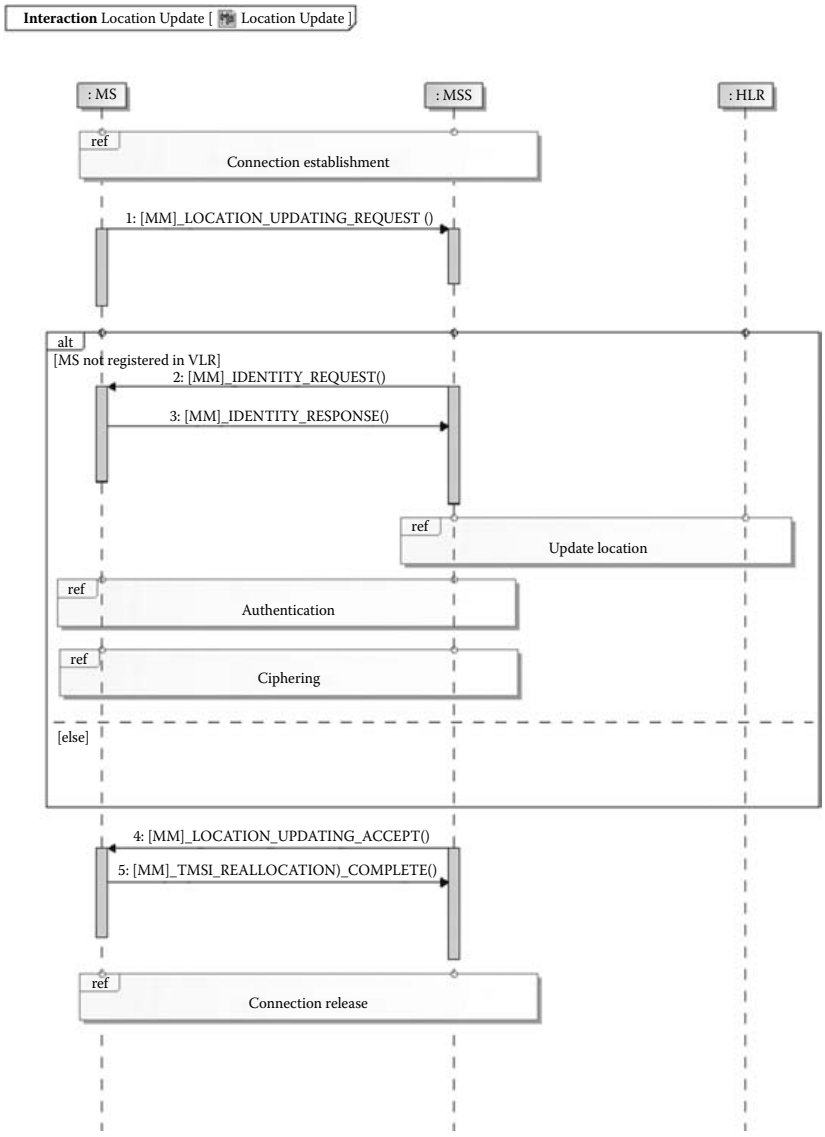


FIGURE 17.10
Sequence diagram describing the location update procedure.

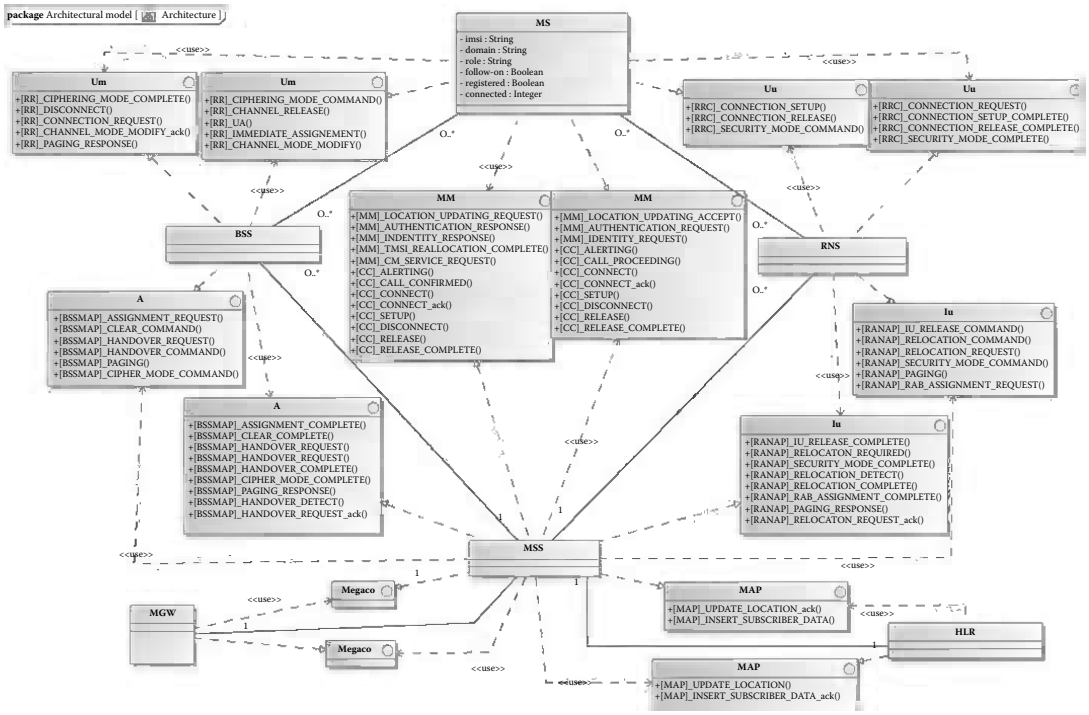


FIGURE 17.11
Class diagram depicting the system domain. (Reproduced from Malik, Q. A., Jääaskeläinen, A., Virtanen, H., Katara, M., Abbors, F., Truscan, D., and Lilius, J., Using system models vs. test models in model-based testing. In *Proceedings of "Opiskelijoiden minikonferenssi" at "Tietotekninen tuki ohjelmoinnin opetuksessa,"* © 2009 IEEE.)

`[MM]_` prefix depicts a class method used at the MM protocol. The set of messages that can be exchanged between the MSC Server (MSS) and the Mobile Subscriber (MS) at the MM protocol are modeled by two different class interface elements (i.e., MM), one for each direction of communication. The approach allows a clear separation of the communication between different network elements and on different protocol levels. The domain model is built iteratively and is mainly derived from the sequence diagrams and the architectural requirements. Each lifeline in the sequence diagrams generates a new class, whereas the interfaces of the class are obtained from the messages that each lifeline in the sequence diagrams receives.

The *data model* (Figure 17.12) describes the different message types used in the domain model. That is, every message on each interface in the domain model is linked to the corresponding message in the data model. The messages are modeled explicitly via class diagrams. Since, in the telecommunication domain, the main unit of data exchanged between entities is the *message* (or PDU), we focus our attention on how different message types and their structure (*parameters*) can be described. By analyzing the communication requirements and the domain models (where messages have been described as class operation with parameters), we create a data model of the system in which each message type is represented by a *class*, while the parameters of the message are represented as *class attributes*. We structure the message definition based on their corresponding protocols (one diagram per protocol) and use inheritance to model common parameters for a given message. Figure 17.12 shows a class diagram specifying the messages used by the MM protocol. The `MM` super-class defines the parameters common to all messages, whereas leaf classes define mandatory parameters for each message type. Optional parameters can be added following a similar approach. One important aspect of the data model is that it does not contain all the mandatory fields of a given PDU, but only those necessary to model the SUT at the current abstraction level. The rest of the parameters will be set up during the test generation level when the abstract test cases will be transformed into executable ones (see Section 17.5, Test Concretization).

State models are used to describe behavior of the SUT. The state model of the MSC Server is derived by analyzing sequence diagrams of each use case one by one. That is, for

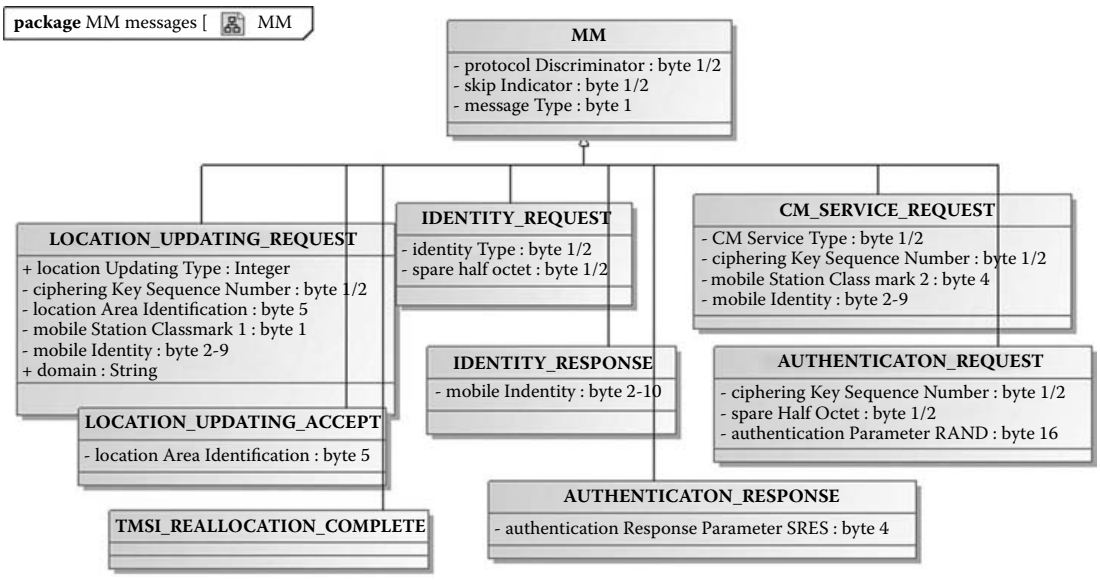


FIGURE 17.12
Data model, defining the structure of MM protocol messages. (Reproduced from Abbors, F., Backlund, A., and Truscan, D., MATERA—An Integrated Framework for Model-Based Testing, *Proceedings: 2010 17th IEEE Conference and Workshops*, © 2010 IEEE.)

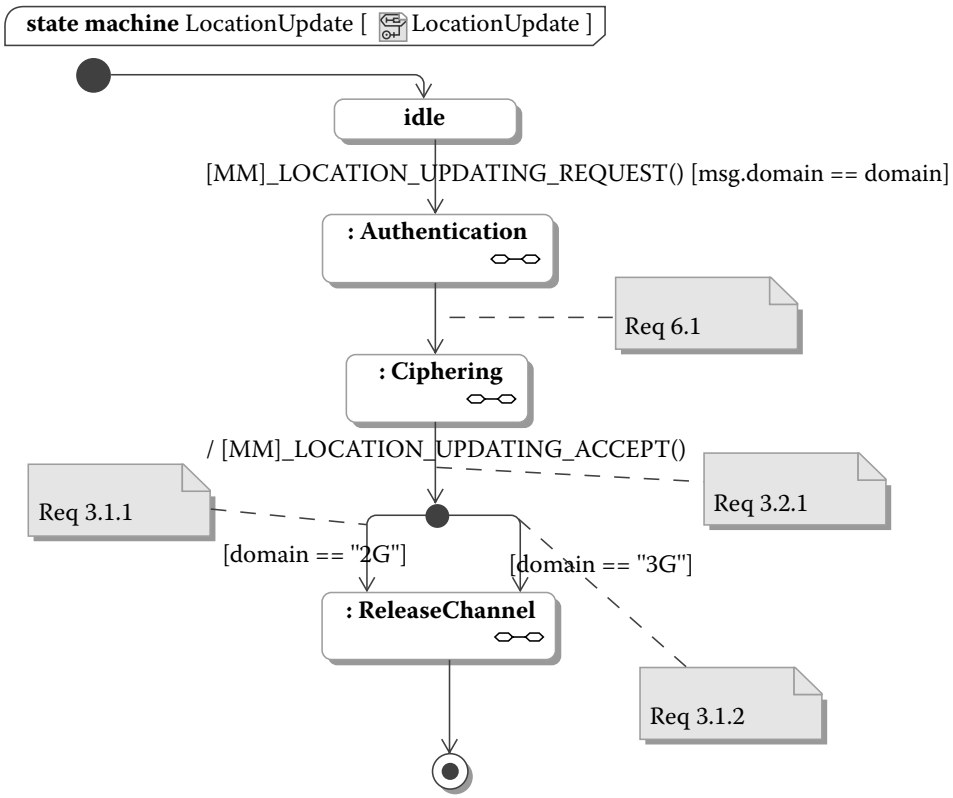


FIGURE 17.13
State machine diagram describing the system behavior.

each sequence diagram, the state of a given object in which each message is received and what messages are sent from that particular state is identified. The former will become *trigger messages*, while the latter will become *actions* on the state machine transitions. By overlapping the states and transitions extracted from each sequence diagram, the full state model of the SUT is obtained. The resulting state model may also contain hierarchical states that will help in reducing the complexity of the model. Figure 17.13 shows a state machine model of the SUT for the location update procedure.

In the *last phase*, a *test configuration model* is created. This model is represented using a UML object diagram and serves to specify the test setup. The elements of this diagram are basically instances of the entities defined in the domain model, showing a particular configuration at a particular point in time. Figure 17.14 shows a test configuration with two mobile phones connected to a 2G network and a 3G network, respectively.

17.3 Model Validation

Humans tend to make mistakes and omit things. Therefore, in modeling, it is necessary for the models to be validated before using them to, for example, automatically generate code or test cases. In our methodology, we take advantage of the model validation functionality of MATERA to check the models for consistency, correctness, and completeness before proceeding to the next step of the process. The idea behind consistency validation is to check for contradictions in the model, for example, a message name in a sequence diagram should match the name of the operation in the class corresponding to that lifeline. Correctness ensures that the models conform to the modeling language (e.g., UML), whereas completeness checks that all necessary information fields have been properly filled out for each

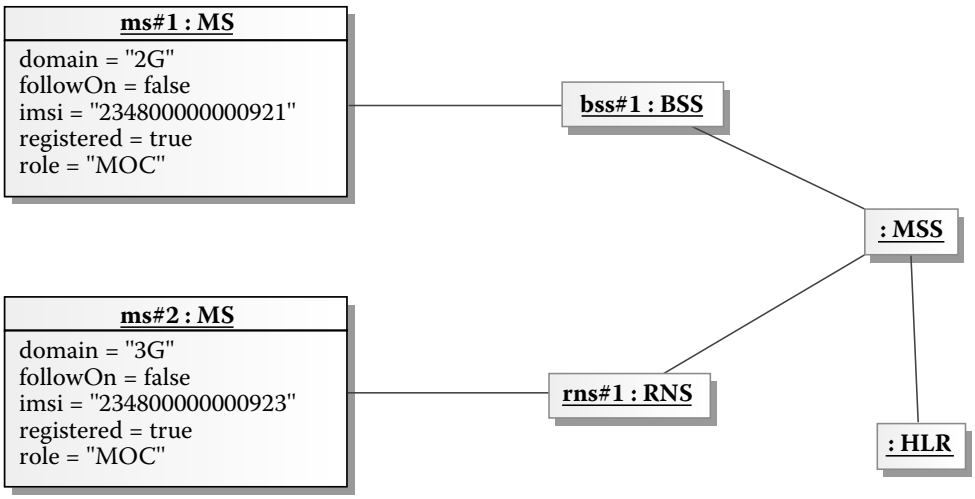


FIGURE 17.14
Test configuration model with two mobile phones connected to the SUT.

element. Model validation can be considered "best practice" in modeling since it increases the quality of the models by ensuring that all relevant information is present or dependencies between elements are correct.

The MATERA framework utilizes the validation engine of MagicDraw for model validation. The engine uses the OCL (Object Management Group b), a formal language for specifying rules that apply to UML models and elements. These rules typically specify invariant conditions that hold true for the system being modeled. Rules written in OCL can be checked against UML models and it can be proved that nothing in the model is violating them. UML, the main modeling language supported by MagicDraw, is accompanied by several predefined suites of validation rules. For example, UML models can be validated for correctness and completeness. There are also additional validation suites for different contexts that can be used. For example, in the SysML context there is a set of validation rules that apply to SysML diagrams. The different validation suites can be checked against either all models or selected models.

Beside predefined validation rules provided by MagicDraw, custom (domain-specific) validation rules can also be created and executed, using the MATERA framework. A complementary set of validation rules was defined (Abbors 2009b) in order to increase the quality of the models with respect to the modeling process. The main purpose of these rules is to ensure a smooth transition to the subsequent steps in the testing process such as generating the input specifications for the test generation tool, or the test generation itself. Next, the validation suites in MATERA and the creation of custom validation rules are described.

An OCL rule normally consist of three parts: (1) a *context* that defines to which language elements the rule applies (e.g., class or state), (2) a *type* that specifies if the rule is a, for instance, an invariant, a precondition, or a postcondition, and (3) the *rule* itself. Optionally, an OCL rule can also contain a *name*. An example of an OCL rule is shown in Listing 17.1.

Listing 17.1
Example of an OCL rule

```
context Region inv initial_and_final_state:
subvertex → exists(v:Vertex|v.oclAsType(Pseudostate).kind =
PseudostateKind::initial) and subvertex → exists(v:Vertex|
v.oclIsTypeOf(FinalState))
```

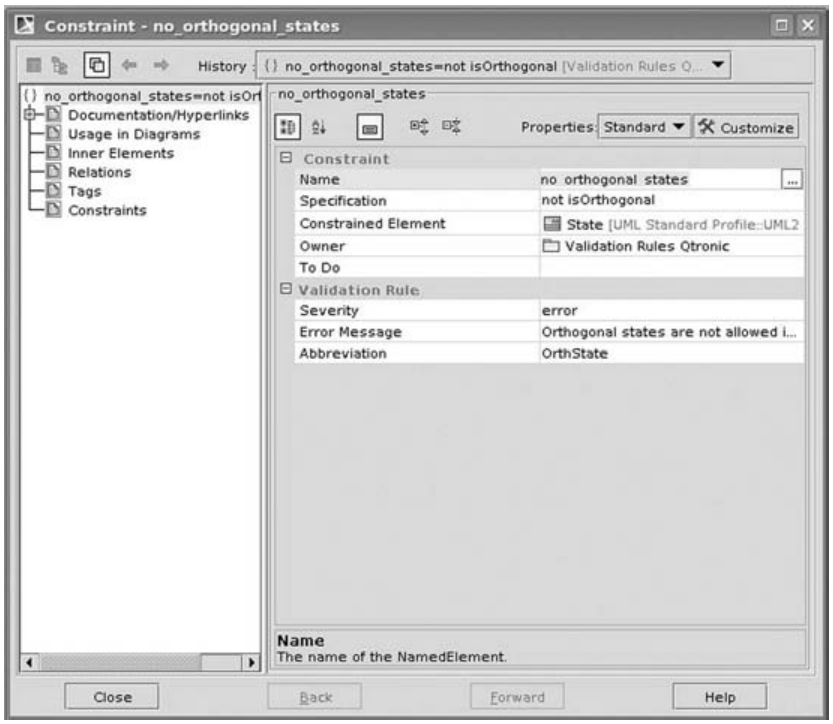


FIGURE 17.15
Screen shot of MagicDraw showing the properties of an OCL rule.

The OCL rule in this example checks that every state machine has an initial and a final state. As one can see, the context of the rule in this example is a `Region` and the type of the rule is *invariant* (`inv`), which means that the expression must always hold true for every instance of type Region. In this example, the OCL rule also has a name, "initial_and_final_state," specified next to the type. By assigning proper names to OCL constraints, it becomes easier to understand the purpose of the constraint and, in addition, it allows the constraint to be referenced by name.

In the validation engine, OCL constraints are treated similarly to model elements. As a result, each constraint has a number of editable properties such as *name*, *specification*, *constrained element*, etc. (see Figure 17.15). The *name* property specifies the name of the constraint. The specification property specifies the rule itself and its type, while the constrained element specifies the context of the constraint.

The validation suites can be invoked at any time during the model creation process. Upon invocation, each rule will be run against the element types or element instances for which it has been defined. If elements violating any rule are found, the user is notified in a Validation Results editor (Figure 17.16). By clicking a failed rule, the elements violating the rules are presented to the user.

17.4 Model Transformation—From UML to QML

The resulting collection of UML models is used for generating the input model for the Conformiq's Qtronic test generation tool, using the MATERA model transformation module. The model in Qtronic is specified using the QML which is discussed more in Section 17.5.1. QML is a mixture of UML state machines and Java, the latter being used as an action

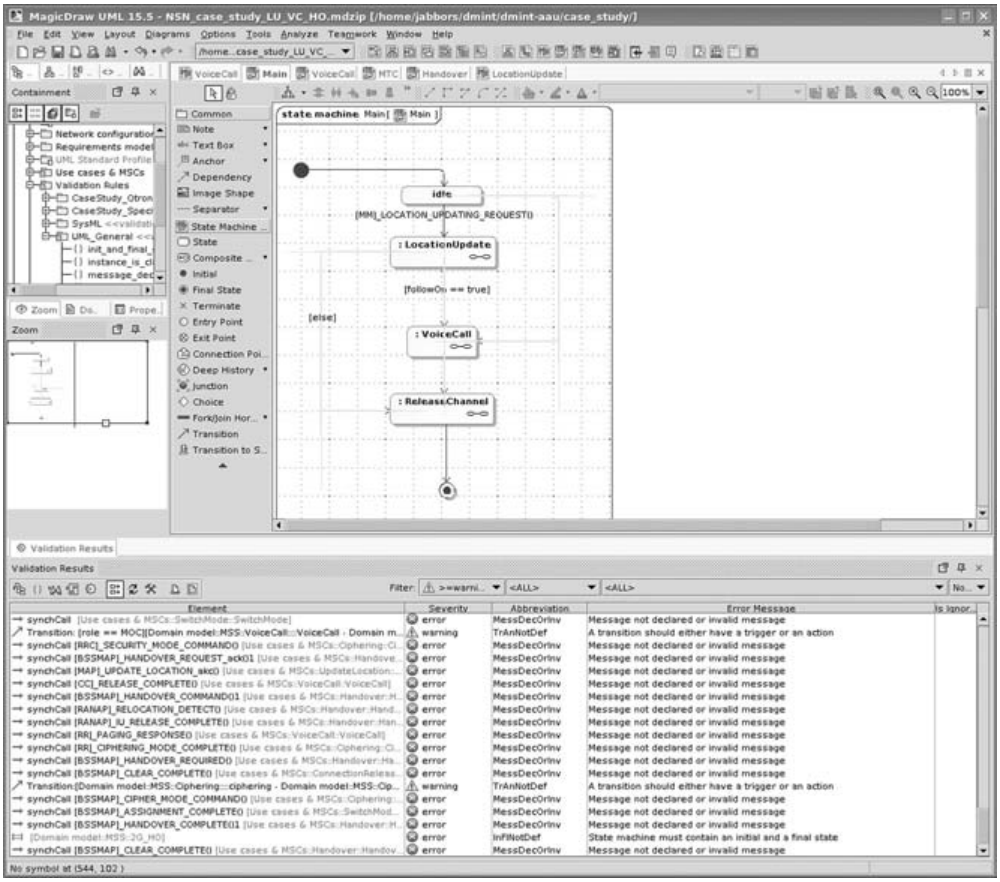


FIGURE 17.16
Screenshot of MagicDraw showing the Validation Results editor. (Reproduced from Abbors, F., Backlund, A., and Truscan, D., MATERA—An Integrated Framework for Model-Based Testing, *Proceedings: 2010 17th IEEE Conference and Workshops*, © 2010 IEEE.)

language. As such, the system models created in MagicDraw are not directly compatible with the Qtronic tool and hence must be transformed into a representation understood by Qtronic, namely QML. The MATERA model transformation module (Abbors et al. 2009) automatically transforms the UML models into the corresponding QML representation. Figure 17.17 shows how different models are mapped onto QML.

17.4.1 Generating the interfaces and ports of the system

In QML, interfaces are specified within a `system`-block. The interfaces describe the ports that can be used to communicate with the environment and which message types can be sent and received on each port. The `Inbound` ports declare messages to be received by the SUT from the environment, whereas the `Outbound` ports declare messages to be sent from the SUT to the outside world.

The ports of the SUT are obtained directly from interface classes in the domain model (see Figure 17.11). `Inbound` messages are taken from the interface realization offered by the SUT, and `Outbound` messages are taken from the interface realization used by the SUT. The name of the ports will be composed of two components, the direction and the interface name. UML operations are listed as messages that are transferred through the ports. The structure of the messages is declared elsewhere as `records`. The partial result of applying the transformation on the `MM` interfaces in Figure 17.11 is shown in Listing 17.2.

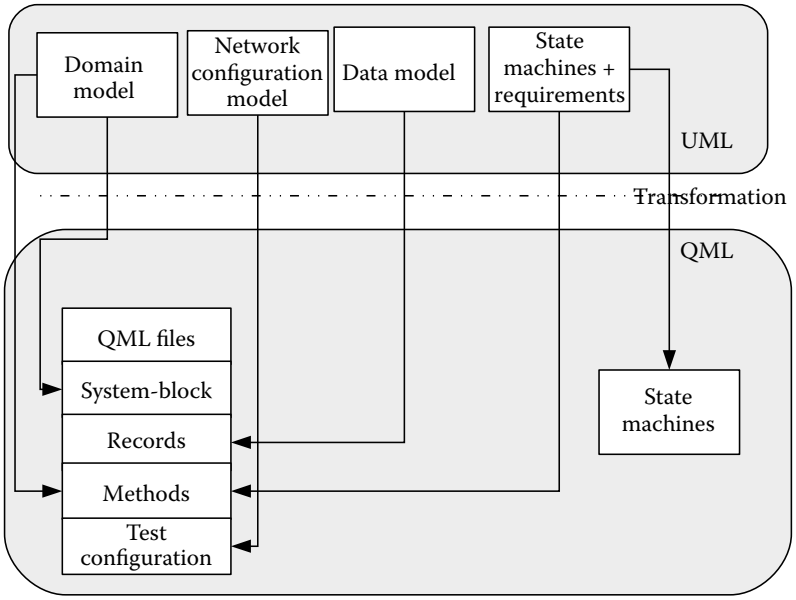


FIGURE 17.17
Mappings from UML to QML.

Listing 17.2
An example of generated Example of QML `system`-block

```
//System block example
system {
  Inbound MM_in: location_updating_request,
      authentication_response, identity_response,
      TMSI_reallocation_complete, CM_service_request, alerting,
      call_confirmed, connect, connect_ack, setup, disconnect,
      release, release_complete;
  Outbound MM_out: location_updating_accept,
      authentication_request, identity_request, alerting,
      call_proceeding, connect, connect_ack, setup, disconnect,
      release, release_complete;
}
```

17.4.2 From UML data models to QML message types

In QML, messages are described as records that are used for communicating with the environment. QML records are user-defined types similar to classes. The fields of a record can be of type: `byte`, `int`, `boolean`, `long`, `float`, `double`, `char`, `array`, `String`, or of another record type. In the transformation, records are obtained from classes in the UML data model. Attributes of the UML classes are transformed into the fields of the record. Inheritance in UML is reflected in QML using the `extends` relationship. For instance, the `location_update_request` record in Listing 17.3 is obtained from a class with the same name in Figure 17.12 following the described approach. The model does not indicate value ranges of the fields. Instead, the value ranges can be checked by the protocol codecs provided by the test system.

Listing 17.3
QML record declaration for LOCATION_UPDATING_REQUEST

```
record MM_messages{
  public String protocol_discriminator;
  public String skip_indicator;
  public String message_type;
}
//record inheritance
record location_updating_request extends MM_messages{
  public int location_updating_type;
  public String ciphering_key_sequence_number;
  public String location_area_identification;
  public String mobile_station_classmark_1;
  public String mobile_identity;
  public String domain;
}
```

17.4.3 Mapping the UML state machine to the QML state machine

As mentioned previously, the behavior of the SUT can be specified in Qtronic either textually in QML or graphically using a restricted version of the UML state machines. For simplicity, the latter option was chosen as the target of our transformation. Thus, the transformation is basically a matter of transforming the UML state machine into the corresponding state machine used by the Qtronic tool, which in practice is equivalent with a transformation at the XMI-level. Figure 17.18 shows the same state machine transformed to QML. As one can see, there is a strong similarity between the two models, albeit with small differences. For instance, both state and substate machines are supported and propagated at the Qtronic level. In UML, triggers and actions are declared as methods (selected only from the operations of the interface classes in the domain model).

In QML, triggers are implemented by messages (record instantiations) received on a certain port, whereas actions can be seen as methods of the SUT class definition. This approach allows one to perform further processing of the system data before sending a given message to the output port.* The method generated for the `MM__LOCATION_UPDATING_ACCEPT()` in Figure 17.11 is shown in Listing 17.4.

Listing 17.4
Example of a generated QML method

```
void MM_LOCATION_UPDATING_ACCEPT(){
        MM_LOCATION_UPDATING_ACCEPT location_updating_accept
            ;
        MM_out.send(location_updating_accept);
        return;
    }
```

*If necessary, additional QML instructions can be manually inserted into the generated methods, before the `.send()` statement.

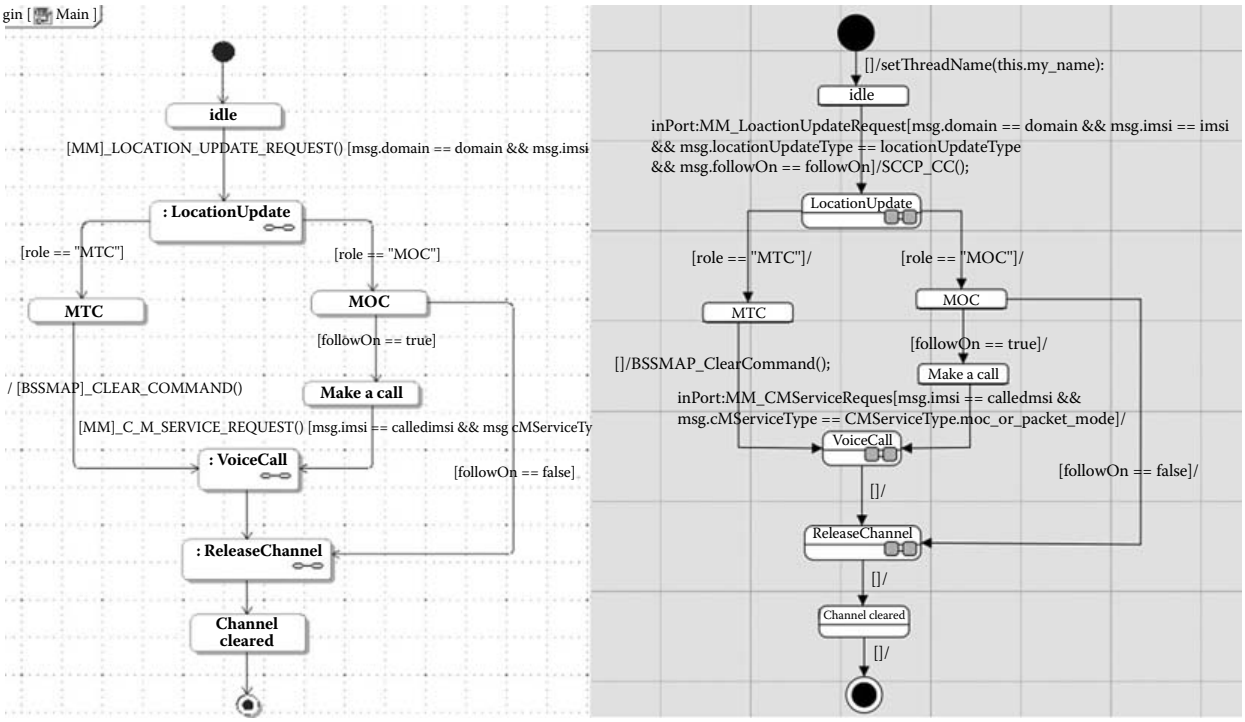


FIGURE 17.18
Example of a UML state machine in MagicDraw (left) and its equivalent in QML (right).

17.4.4 Generating the QML test configuration

As illustrated in Figures 17.11 and 17.14, a `:MS` is a user of the SUT who communicates with it via the `MM` interface. The test configuration in our example consists of two MS. In QML, a subscriber is modeled as a record (Listing 17.5) with the same attributes as in the domain model (Figure 17.11). The test configuration is translated into QML in two steps. In the first step, the properties of the test components are extracted from the UML domain model and are declared in the constructor method of the SUT specification class (`MSS` in our case) as shown in Listing 17.6. In the second step, each test component is instantiated and the properties are initialized with the values taken from the configuration diagram (Listing 17.7). The test components are stored in an array, which is later used for starting concurrent test components of the test system in separate threads.

Listing 17.5
Example of subscriber record

```
record Subscribers {
        public String my_name;
        public boolean followOn;
        public String domain;
        public String role;
        public boolean registered;
        public String imsi;
        }
```

Listing 17.6
Example of subscriber record initialization

```
public InitializeSubscriber(Subscribers sub){
        my_name = sub.my_name;
        followOn = sub.followOn;
        domain = sub.domain;
        role = sub.role;
        registered = sub.registered;
        imsi = sub.imsi;
        }
```

Listing 17.7
`main` method of the SUT specification class

```
// *** MAIN ***
void main(){
        Subscribers mySubscribers[] = new Subscribers[3];
        mySubscribers[0].my_name = "ms#1";
        mySubscribers[0].followOn = false;
        mySubscribers[0].domain = "2G";
        mySubscribers[0].role = "MOC";
        mySubscribers[0].registered = true;
        mySubscribers[0].imsi = "234800000000921";

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

        for(int i = 0;i<=1; i++){
            MSS mss = new MSS(mySubscribers[i]);
            Thread t = new Thread(mss);
            t.start();
        }
}
```

17.4.5 Assigning the state model to the SUT specification

At this point, the only thing that remains to be done is to connect the SUT class specification to the graphical state model. This is done by calling the constructor method for the state machine. Once the state machine has been constructed, the concurrency of the SUT can be tested by starting (via the `Thread.start()` method) separate execution threads for each test component (i.e., subscriber) (Listing 17.8). The approach allows for different MSs to concurrently communicate with the SUT using different configuration parameters. This is necessary as in telecom systems such as the MSC Server, one must test the presence of

multiple MSs interacting with the MSC Server (in practice, one MSC Server can serve up to several million users). In addition, we must test calls between pairs of subscribers, where one call requires two subscribers, that is, the caller and the receiver (known as A and B subscribers).

Listing 17.8
State machine instantiation in Qtronic

```
for(int i = 0;i<=1; i++){
            MSS mss = new MSS(mySubscribers[i]);
            Thread t = new Thread(mss);
            t.start();
        }
```

17.5 Test Generation

The test generation phase follows the modeling phase in the overall process. The test generation exploits the QML models transformed from the UML models. The supporting tool chain implements the test generation phase using the Conformiq Qtronic tool. Qtronic is a tool for Automated Test Design. It derives tests automatically from system models that represent the desired behavior of the SUT. The generated tests are black-box tests and so they evaluate the SUT based on its external behavior, not by monitoring its internal workings.

17.5.1 QML

The systems model given as input into Qtronic is expressed in terms of a language called QML. A model is a collection of the following:

- Textual source files in a Java-compatible but extended notation that describe data types, constants, classes, and their methods.
- UML state-chart diagrams representing the behavioral logic of active classes as an alternative to representing the logic textually.
- Class diagrams as a graphical alternative to declare classes and their relationships.

A QML model is therefore essentially an object-oriented computer program, an abstract implementation of the system to be tested.

The diagrams can be drawn using various tools that Qtronic works with, such as Conformiq Modeler, Enterprise Architect, IBM Rational Software Developer, or IBM Rhapsody. It is also possible to create models completely textually, that is, all the diagram types are optional.

17.5.2 Test generation criteria

Given a system model, Qtronic automatically identifies a number of test cases that together cover the testing goals selected for test generation. Appropriate test input data as well as the correct expected output is automatically calculated and generated by the tool without further input from the user.

For this, Conformiq Qtronic uses semantics-driven methods for generating test suites, which means that test generation is guided by an analysis of the behavior implied by the model, instead of being based on syntactic analysis or simple heuristics. Qtronic uses model-based coverage criteria to select a set of test cases to form a good test suite. The coverage or testing goals are used to guide Qtronic to look for certain behaviors from models or to enable certain behaviors.* A test case covers a certain testing goal if execution of the test against the model itself would cause the goal to be exercised. Then, Qtronic uses its capability to simulate the system model to construct test cases, and at the same time, it maps the test cases to the different test goals that result from the coverage settings. It then selects from the constructed test cases a set that covers all of the resulting test goals using a minimal cost test suite. This ensures that the suite is reasonably small and compact, and at the same time, the individual test cases remain relatively short, which eases test execution and debugging. In addition to this, Conformiq Qtronic also prefers covering all test goals as early as possible, that is, after as few messages as possible, providing better separation of concerns between test cases.

A test suite generated by Qtronic has the following characteristics:

- In order to have good error detection capabilities, the generated test suite covers as many testing goals as possible.
- In order to avoid redundant testing, the generated test suite is as compact as possible while individual test cases in the suite are relatively short in order to ease the test execution and debugging.
- In order to provide better separation of concerns between test cases, the test goals are covered as early as possible in test cases.

Qtronic makes the testing goals accessible to the user in the Qtronic user interface (UI). By selecting different testing goals, the user can affect how Qtronic generates test cases. This is the primary vehicle in Qtronic for a user to have a say in how the tests are generated. Figure 17.19 shows testing goals in the Qtronic UI. From this view in the coverage editor the user can see, for example, that the generated test suite covers all requirements (more on requirements below) related to category "1.2" but not all requirements in some other categories. One can also see that 62% of all the states and 60% of the transitions are covered by the test suite.

The selection testing goals that are used depend on how extensive a test suite the user wishes to generate and also on the characteristics of the model and hence the SUT itself. For example, if the model has lots of interesting boundary conditions related to inputs to the system, it is highly recommended for one to enable Boundary Value Analysis as a test generation criterion (see Figure 17.19). With boundary value analysis enabled, Qtronic will attempt to exhaustively generate all possible boundary values based on the various conditions in the model.

Not all testing goals are always reachable. One can, for example, have conflicting if-statements in the model, which cause a certain boundary value case to be statically unreachable. An important aspect of how Qtronic works is that Qtronic will give the user a precise account of which coverage goals were covered by the generated test suite and which ones were left uncovered. Therefore, after test generation, the user knows exactly how well the generated test suite covers the different functionalities of the modeled system (and can react to uncovered areas and change the model if it turns out that there was a defect in the model itself).

*Both of the terms, *coverage goal* and *testing goal*, are used in this chapter and they mean the same thing.

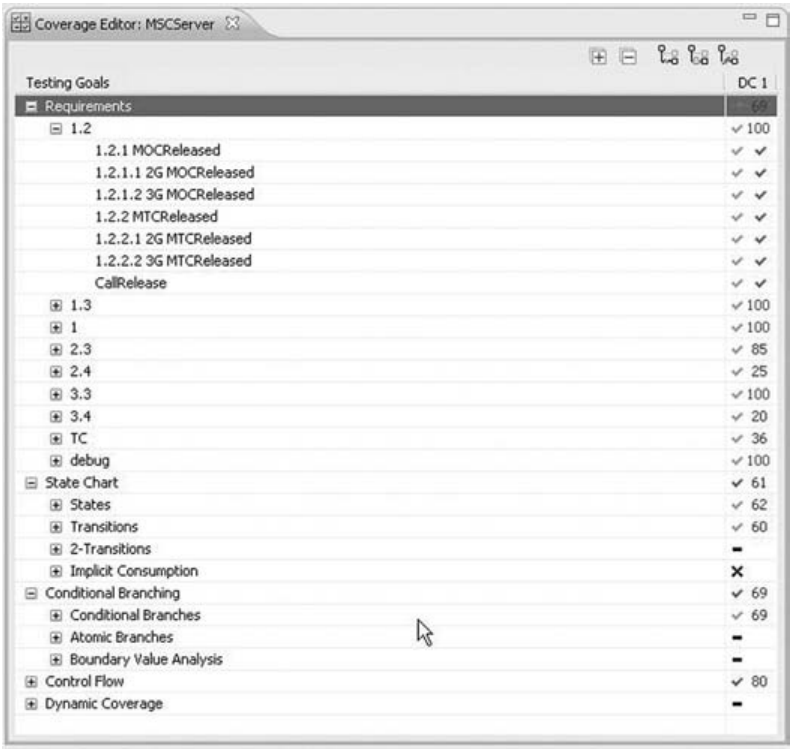


FIGURE 17.19
Testing goals in the Qtronic user interface.

17.5.3 Requirements traceability

Most of the testing goals are related to various properties of the model itself (such as state, transitions, conditional branches, etc.). In addition to these structural coverage criteria, user-defined requirements can be used in models to guide the test generation.

Technically, the requirements are embedded in the model using the requirement keyword, as illustrated in Listing 17.9.

Listing 17.9
An example of a requirement in a model

```
// A requirement embedded in a model
requirement ''This is a requirement'';
```

For the Qtronic algorithm, a requirement is one additional coverage goal in the model to be covered (see discussion on testing goals in Section 17.5). In generating tests, Qtronic will attempt to generate a test suite that covers all requirements embedded in the model at least once, assuming that the user has chosen requirements as a testing goal prior to starting test generation. The notion "covering a requirement" means that there is an execution in the model that passes through the point where the requirement is defined.

At the end of test generation, Qtronic will report how well the requirements in the model were covered by the generated test suite (just like it does for all other testing goals). What is particularly interesting about the requirements coverage is that it provides automatic traceability from the functional requirements of the system all the way through to the generated test cases and test execution. A traceability matrix maps requirements to test cases, allowing a tester to easily pick up test cases to exercise certain functional areas identified by the corresponding functional requirements. An example of a traceability matrix can be seen in Figure 17.20. For example, if the user wishes to execute a test case that

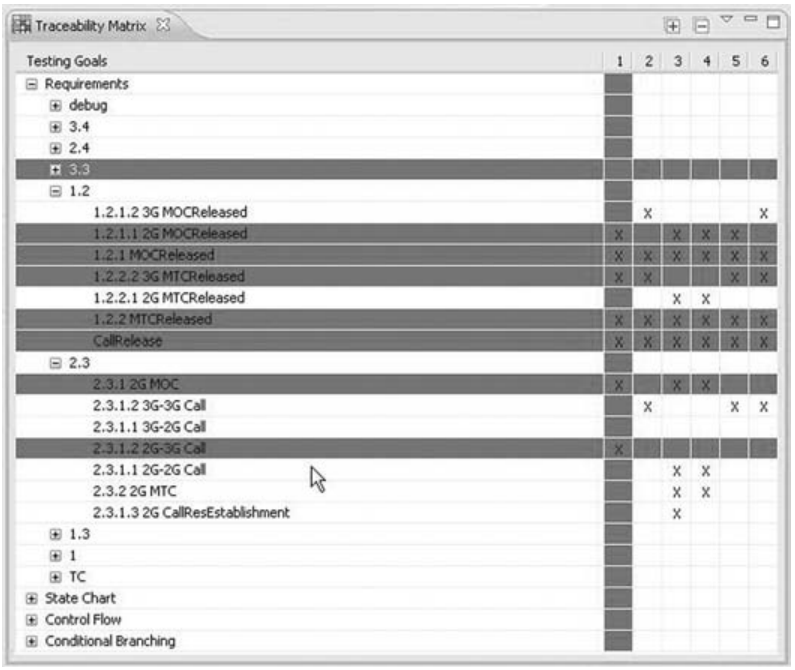


FIGURE 17.20
Traceability matrix in the Qtronic user interface.

exercises a "mobile-originating call from a 2G network to a 3G network," then test case 1 would have to be executed as can be seen from the "Xs" in the highlighted column for test case 1. Furthermore, a failure in test execution can be easily attributed to a functional area by looking at the traceability matrix.

17.5.4 Test concretization

Once the QML system model is created and tests are generated, we have a set of valid test artifacts in terms of test cases, messages, parameters, and testing goals. Test artifacts, from the Qtronic database, must be transformed into an executable format for test execution.

NetHawk's Environment for Automated System Testing (EAST) [3] is used for test execution. EAST embeds the SUT in a virtual network incorporating protocols over interfaces under test. EAST provides a Test Creation Environment (TCE) with graphical programming language for defining tests, test suites, message templates etc., and a Test Execution Environment (TEE) for test execution. In the project described in this chapter, TCE is utilized for creation of the message reference library. TEE is used for executing test cases in Load Testing mode, which provides an execution performance close to real network elements. EAST TEE connects a test execution engine to the desired protocol server required by a protocol under test. A Protocol server is a standalone program simulating a protocol stack below the protocol under test. In this context, *Protocol under test* refers to the level of the modeled behavior of the SUT.

The message reference library is a collection of message descriptions and encoding rules of telecom *protocols under test*. In other words, it is a definition of the run-time behavior of messages. EAST provides most Protocol Data Units (PDU) of telecom protocols in the form of message templates with default content. Each value of the template field in the reference library is accessible through an API. It is up to the user to select the necessary

PDUs from the reference library and create mappings between the parameters from test artifacts and reference library messages. Data models, depicted in Figure 17.12, define the interfaces under test on an abstract level when the message reference library is the actual implementation of PDUs. One could think of the reference library as a refinement model for the data model.

For example, the data model defines a message *MM_LocationUpdateRequest* and Qtronic has generated test artifacts defining parameters and behavior of when the message is sent by the test system. In an executable test case, the event of sending the *MM_LocationUpdateRequest* message will be composed from the implementation of the message in the reference library and of test artifacts calculated by Qtronic. Figure 17.21 shows the relation between reference library and generated test artifacts.

In order to concretize such an executable test case, it is required that there be a tool to combine Qtronic test artifacts and the message reference library refinement data producing EAST executable test cases. For this purpose, a test scripting back-end was implemented. The back-end is connected to Qtronic using an open API. Through the open API, it is possible to create a custom output format (e.g., EAST scripts) from test artifacts and utilize external test libraries such as the EAST reference library. The back-end creates a set of test cases that can be executed on EAST TEE.

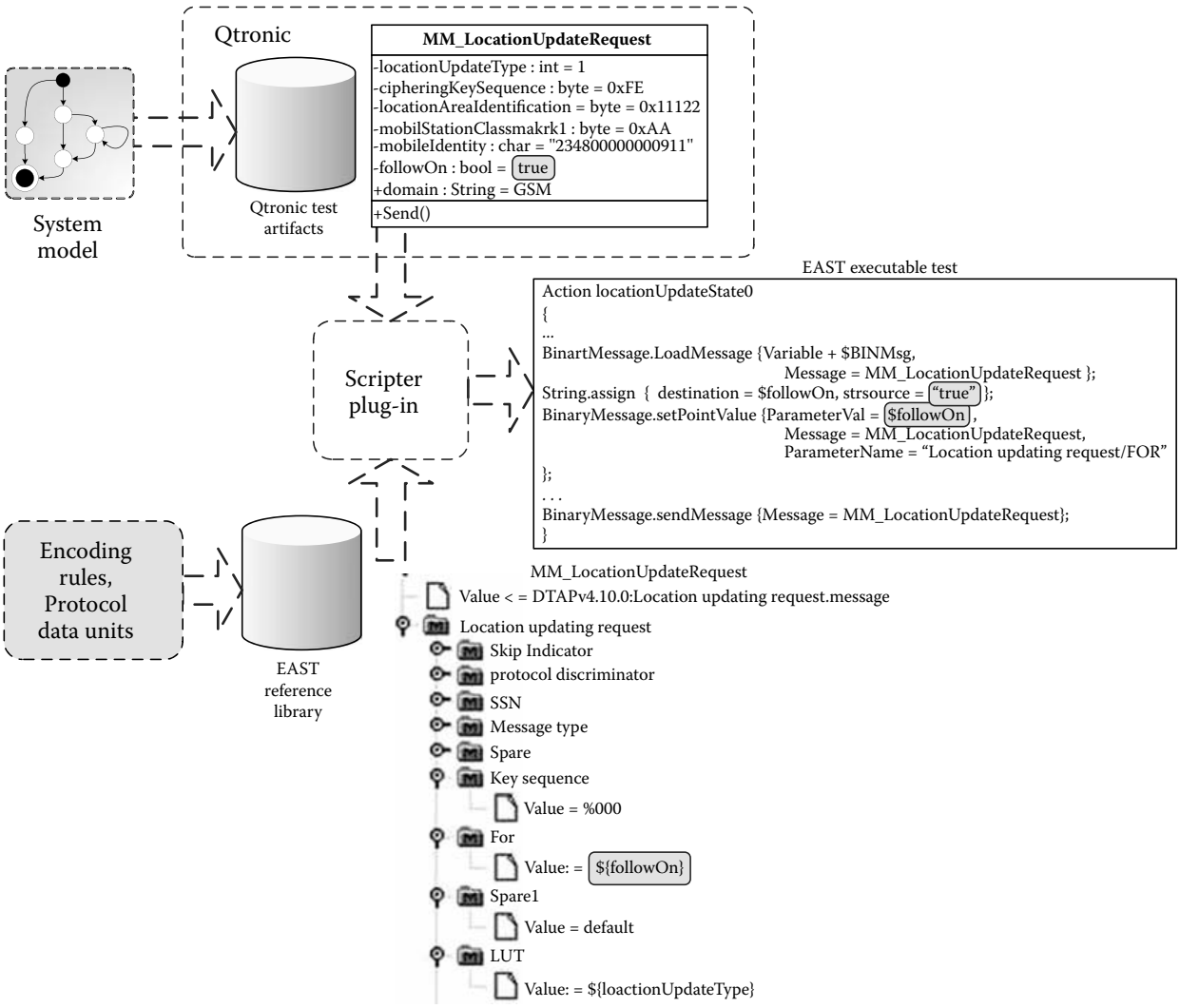


FIGURE 17.21
Test concretization.

The message reference library may have several versions supporting several protocol versions. All versions implement the same API in order to keep maintenance work to a minimum. The API defines the mapping between the reference library messages and the abstract definitions of messages. If the specification version of the protocol under test changes, the reference library version also must be changed. Abstract definitions and the data model should not have to change.

17.6 Test Execution

Test execution is performed with the help of a test system. The test system is illustrated in Figure 17.22. The test system operation is automated with a custom-made Test Automation Script (TAS). The test system is based on the NetHawk EAST Test Execution Environment (TEE) and on *protocol servers* providing connectivity to the SUT. In addition, the message reference library, described in Section 17.4, is providing run-time behavior for PDUs called from EAST test scripts. EAST produces test logs on each test run. The logs are used in the analysis and for coverage tracing purposes later on as will be discussed in Section 17.6. The communication between the MSC Server and the test system is monitored with the Wireshark (Wireshark) protocol analyzer. The message reference library was discussed earlier in Section 17.4 from the test concretization point of view. During the test execution, the message reference library is providing an API for encoding and decoding rules for messages sent and received. For example, if a message *MM_LocationUpdateRequest* is about to be sent by the TEE, the message reference library provides information about online encoding of the message.

The TAS is used for starting related test steps at the same time, for collecting combined coverage results, and for calculating the final verdict of the test case. After the system model is created and test generation performed, the TAS takes care of all steps required in the test execution phase.

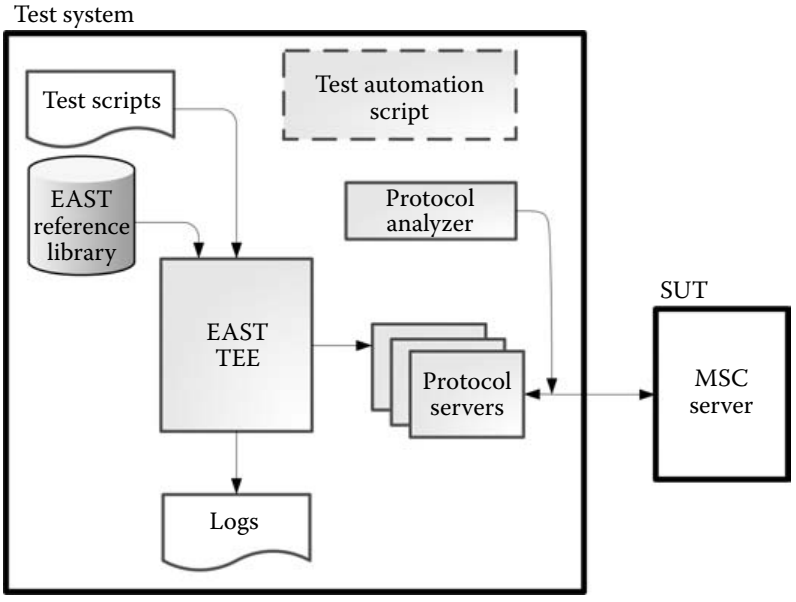


FIGURE 17.22
Architecture of the test system.

17.6.1 Load testing mode

The Load Testing mode has been selected from different execution environments provided by EAST TEE. The Load Testing mode provides an execution engine allowing several simultaneous calls and has a means for identifying different calls by a special Context. The Context in our environment defines a sequence of actions or events having certain unique characteristics. For example, when simulating an originating phone call, the calling party has its own phone number and the called party has its own number. Both, originating and terminating, calls have their own Context that can be addressed by a phone number as phone numbers are unique. Also, while a phone call is ongoing, the Context can be referred to at a protocol level through a specific connection in case of a connection-oriented communication.

17.6.2 Concurrency in model-based testing

Modeling concurrent systems, such as telecom network elements, requires some extra attention in the modeling phase in order to obtain executable tests from the test generation. The SUT is concurrent by nature, which means that several sequences of events are happening at the same time without synchronization between them. This leads to a situation where it is possible to have large amounts of different, but correct, interleaved sequences of elementary procedures. For example, Event A (EA) has 10 elementary procedures that always happen in a known order. Respectively, Event B (EB) has 10 elementary procedures. In a case where EA and EB are happening in parallel, there are $2^{20-1} = 524,288$ possibilities for interleaved elementary procedures of EA and EB. In reality, there are some relationships between EA and EB which narrow down the possibilities. But, when looking from a test generation point of view, it is difficult to create information on models that rule out the possibilities that do not take place in reality. And even after that, it is neither feasible nor wise to generate all possible variations. The most important point is to not generate a test that has one fixed order of parallel events. That is because it is not known beforehand in which order these parallel events happen in the SUT.

For example, originating and terminating calls receive a confirmation when they are connected to each other, but it is not specified which phone receives the confirmation first. Some branching would be necessary in a single test case to cover these types of situations. This will lead to a very complex test case that requires a lot of computational power from the test generation tools. Instead of branching, the solution is to treat originating and terminating calls as separate threads and generate independent test steps for both calls. These steps together form a test case. Both steps are increasing the test coverage (see Section 17.4) of the test case while they are running.

17.6.3 Executable test case

MBT test generation tools such as Qtronic can handle only deterministic behavior. An individual test case should go through a deterministic path on each execution round. But as illustrated in the above example, EA and EB as one test case, there are several options on how a valid test case could behave on interleaved execution leading to nondeterministic behavior. At first, this sounds like a restriction from the test generation tool perspective, but when thinking about this dilemma openly, this is the only way that a test generation tool could produce something reasonable. Concurrency must be handled in the test execution environment and taken into account when creating system models. The system model should define a separate thread for both the originating and terminating phone call. The test generation tool produces one test case where EA and EB are combined in one interleaved

sequence of events. In the creation of an executable test in the test concretization phase, described in Section 17.4, executable events of EA and EB are separated on their own test steps. The entire process is depicted in Figure 17.23.

17.6.4 Context

Figure 17.24 depicts the message routing from the SUT to the correct test step in the test system and vice versa. In the picture, the Router is a book keeper of the active Contexts in the test system. It compares the received message type, header, and possible payload to infer the correct Context. The EAST test case from the example above has two test steps with unique phone numbers. There is also a possibility to define a unique protocol transport layer identifier for each test step, for simplicity, is referred to as *connectionID*. Most of the communication between the SUT and the test system is connection oriented and it is easy to determine an identifier, *connectionID*, for each connection during the connection establishment phase. However, because of the nature of the telecom network, there are some connectionless messages exchanged within the network. These messages are mainly used for paging other phones or resources before the connection is known. Here, it is assumed that these broadcast messages carry an address of an entity to which the message belongs. To simplify the example, the address is referred to as *phoneNumber*.

Generated EAST scripts, one script for each test step, are uploaded to the Load Runner. Each test step registers their Context with a run-time database storing the relation between the Context and the test step. In the example Listing 17.10, the Context can be addressed through two context identifiers, *connectionID* and *phoneNumber*.

Listing 17.10
Example of the relation between the Context and the test system

```
String.assign { destination = $connectionID , strsource = "1" }
LoadEngine.setContextId { Variable = $phoneNumber , type = "reference" }
String.assign { destination = $phoneNumber , strsource = "62030614610001" }
LoadEngine.setContextId { Variable = $phoneNumber , type = "reference" }
```

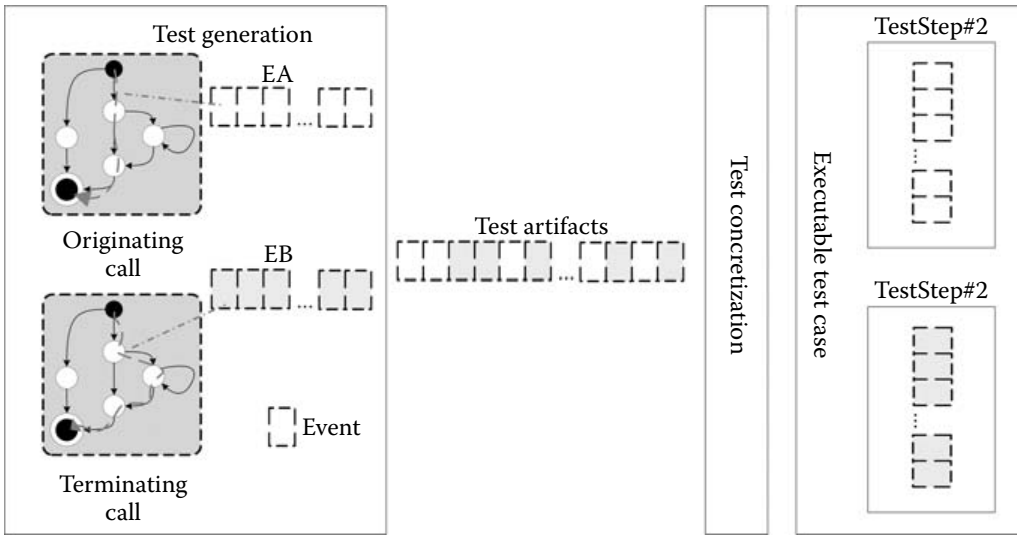


FIGURE 17.23
From the system model to executable test.

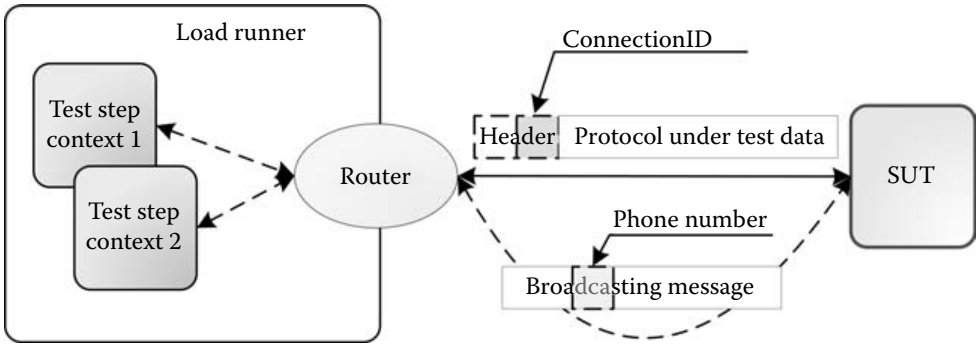


FIGURE 17.24
Context-based message routing.

17.6.5 Run-time behavior

There are two possibilities for the test step to identify its Context. First, it could do a connection establishment after which *connectionID* can be used to address messages through the router. Second, the test step waits for a broadcasting message with *phoneNumber* defining the correct Context after which the connection should be created as in the first case. Broadcast messages are sent to all test steps but discarded if the *phoneNumber* does not match. After the Context is known, the Router depicted in Figure 17.24 knows the destination of each message.

EAST TEE is writing a textual log that contains information of executed events and covered requirements. The log file is an input for the requirement traceability phase described in Section 17.7.

17.7 Requirement Traceability

Conventional test scripting is based on static test cases, that is, a test case has a name and corresponding test code that is not modified significantly between consecutive versions. However, this is not the case when tests are generated from models automatically. In automatic test generation, there are no guarantees that different test generation rounds produce similar test scripts. Thus, as part of the methodology, an approach was developed for tracing the requirements throughout the entire MBT process, from models to test runs and back to models as illustrated in Figure 17.25. This approach provides a consistent way to monitor the modeling, generation, and test execution status, as well as the test completion rates. In addition, tracing of requirements helps understand what kind of impact new or modified requirements have on different artifacts of the testing process. All in all, the requirement tracing process supports short feedback loops that, in turn, support modern product development conventions such as iterative and agile development practices.

The use of the requirements fits well in product testing because the features of the product are defined as a set of requirements. It does not require any additional information on top of the regular product development information. Instead, it uses information that is readily available in the product development. This makes it easier to integrate the MBT tool chain with the current frameworks and with the tools used in product development. In addition, it is also important to be able to evaluate how many tests cover each requirement to support test prioritization and optimization.

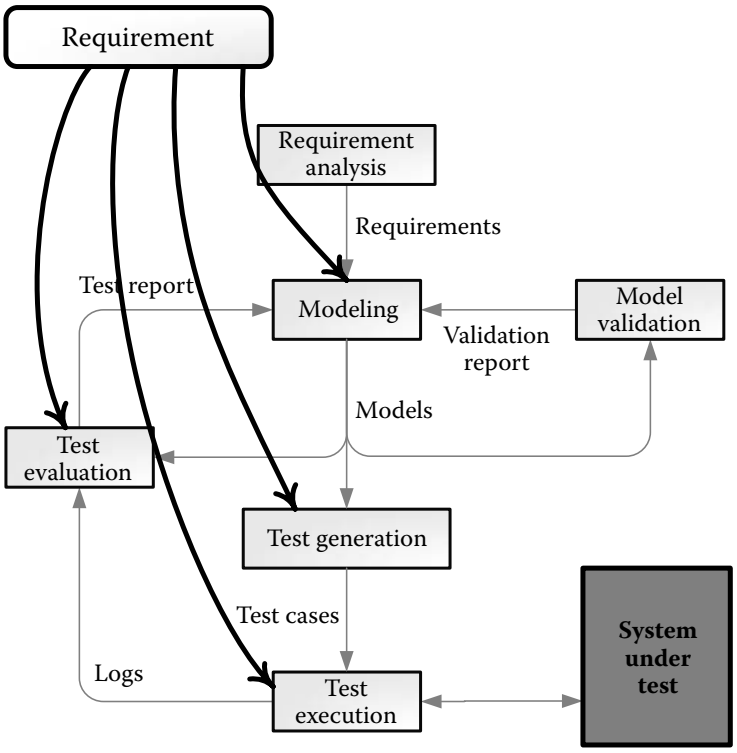


FIGURE 17.25
Tracing requirements throughout the MBT process.

The use of the requirement tracing also helps in the process of test selection and prioritization. Typically in product development, some features (requirements) of the product are more important than others. This affects the testing process in a way that more important features must be tested more extensively compared to others. The prioritization of the test cases can be done in two different phases of the testing process, during test generation or test execution. If the behavioral model of the SUT used for test generation is adorned with priority information, then the test generation tool will be able to generate test cases and calculate their execution order. However, if such a possibility is not supported by the test generation tool, then having the generated test cases tagged with the requirements they are testing will allow the selection of the most important test during execution based on the priority of the requirement. Especially in the telecom domain, where the number of generated test cases tends to be very large, having the possibility to select the most important test cases to be executed based on the requirement to be tested is an important aspect.

At a more detailed level, requirement tracing allows requirements to be propagated to test specifications. Further, functional testing must verify that all requirements have been covered by test. Needless to say, requirements are the keystone in any successful project implementation, and hence, they must be traceable both to models and tests. Tracing requirements during development ensures that all requirements have been implemented and that no functionality has been overlooked. Tracing requirements to tests can even help in identifying missing tests, that is, where critical requirements do not trace to any test. Finally, if a test fails, one can trace the requirement back to the models from where it originated, in order to identify the error. This facilitates the process of identifying which parts of the system model cause a set of test cases to fail. It is described in brief how these aspects are addressed in the following subsections.

17.7.1 Tracing requirements to models

Requirements traceability is one of the key features of MATERA and is based on the approach described in Section 17.2. As presented in Figure 17.7, requirements are structured hierarchically. Top-level requirements are traced to different models (e.g., state machine diagrams), whereas the rest of the requirements are traced to model elements to which they apply (e.g., a transition or a state).

In our modeling process, requirements are propagated to different parts of the models to indicate a relationship between requirements and model elements. For instance, communication-related requirements are traced to data models, architecture-related requirements to architecture models, and functional requirements are initially traced to use case models and then to state models. Figure 17.26 shows an example of how a requirement can be linked to a model element in MagicDraw (e.g., to a transition in a state machine). These links can be useful both for evaluating that all the requirements have been reflected in the models, by showing what elements from different diagrams "implement" a given requirement, and for tracing the requirements to tests. Figure 17.13 presented a partial state model of the SUT in which various transitions have tags depicting requirements that they satisfy.

Once all requirements have been traced to UML elements, validation can be applied to verify that no requirement has been overlooked and that the models are ready for transformation.

17.7.2 Tracing requirements to tests

As mentioned in Section 17.4, Qtronic offers support for tracing requirements during test case generation. MATERA propagates the requirements linked to model elements into QML specifications. During the UML to QML transformation, the requirements are propagated from UML models (namely from state machine models) to QML state machine models. In the current case study, only those requirements that are attached to state transitions are collected from the UML state models. However, nothing prevents the collection of requirements from other UML models.

As such, the requirements are captured from UML transitions and placed on the corresponding transition in QML by using the `requirement`-statement (as explained in Section 17.5). Hierarchy can also be propagated from UML to Qtronic based on the numbering scheme of the requirement.

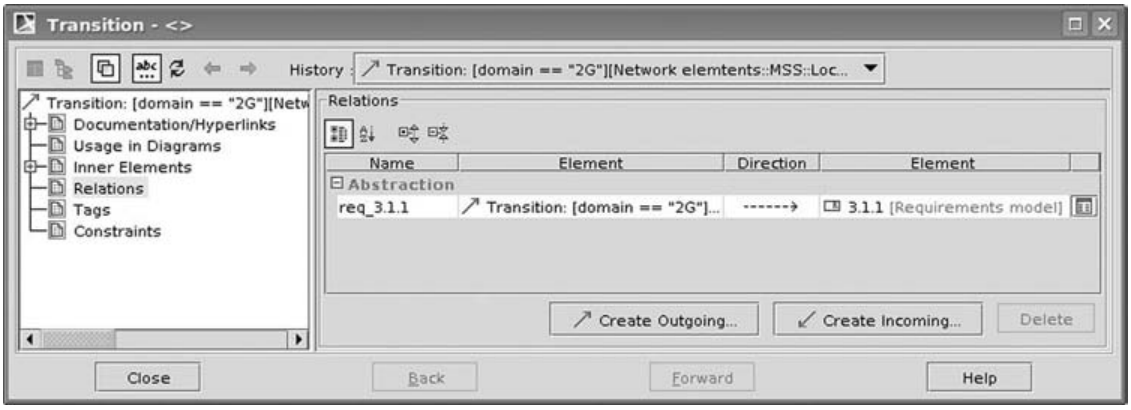


FIGURE 17.26
Tracing requirement to a model element.

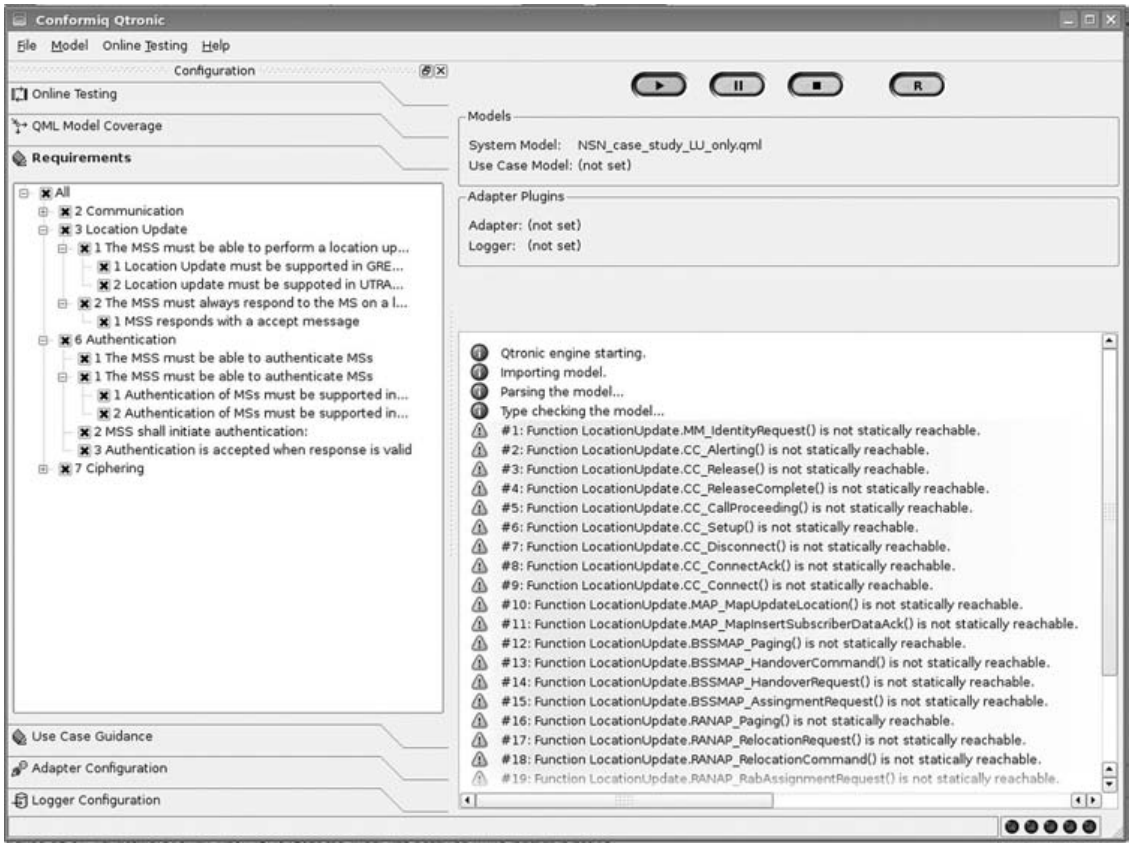


FIGURE 17.27
Example of Qtronic's UI with requirements.

Upon importing a QML model in Qtronic, requirements are displayed hierarchically in Qtronic's UI (Figure 17.27) from where requirements-based test derivation can be pursued. In addition, the user can select which requirements should be covered by Qtronic during test case generation.

During the test case generation, Qtronic propagates the requirements to EAST test scripts via the scripting back-end discussed in Section 17.4. After running the generated test scripts in EAST, the output of the test run is stored in EAST test logs (see Listing 17.11). The test logs contain detailed information on test steps, executed methods, covered requirements, etc.

Listing 17.11
Excerpt from an EAST test log.

```
Log:20
1::1::Symbol: Statement ::Label: Statement :: Script Name: TC_TEST_2G_ms#1_2
    :: When: Tue May 19 14:50:53:451 2009 ::
Assign( ("6 Authentication /1 The MSS must be able to authenticate MSs /1
    Authentication of MSs must be supported in GERAN (2G) networks") )  =
    requirement ( "6 Authentication /1 The MSS must be able to authenticate
    MSs /1 Authentication of MSs must be supported in GERAN (2G) networks" )

Log:21
1::1::Symbol: Statement ::Label: Statement :: Script Name: TC_TEST_2G_ms#1_2
    :: When: Tue May 19 14:50:53:451 2009 ::
```

```
Assign( ("6 Authentication /3 Authentication is accepted when response is
    valid") )  = requirement ( "6 Authentication /3 Authentication is
    accepted when response is valid" )

Log:22
1::1::Symbol: Statement ::Label: Statement :: Script Name: TC_TEST_2G_ms#1_2
    :: When: Tue May 19 14:50:53:451 2009 ::
Assign( ("6 Authentication /1 The MSS must be able to authenticate MSs") )  =
     requirement ( "6 Authentication /1 The MSS must be able to authenticate
    MSs" )
```

17.7.3 Back-tracing of requirements

This approach is the opposite to the ones presented above and it consists of two parts. Firstly, statistical information about the test run regarding the number of passed/failed test cases, number of requirements covered, validated, etc., is collected in a report. The MATERA test evaluation module is used for analyzing the EAST test logs and generating the report (a simplified version of the report is shown in Figure 17.28). During the analysis,

Statistics of the test output from EAST^{tm}

This HTML log has been automatically generated by a script.

The log contains statistical information about test output from EAST as well as requirements coverage information

Generated on: Fri August 14 14:52:47 2009

Requirements Information

Failed Requirements

3.1.1 Location updated must be supported in GREAN (2G) networks

6.1.1 Authentication of MS's must be supported in GERAN (2G) networks

6.1.2 Authentication of MS's must be supported in UTRAN (3G) networks

7.1 The MSS must be able to cipher the communication with MS's

7.2 The ciphering procedure is initiated by the MSS after a successful authentication procedure

Uncovered Requirements

6.1 The MSS must be able to authenticate MS's

Test Case Information

The testrun generated a total of 10 testcases

Requirement 3.1.1 was present in 2 testcases

Requirement 6.1.1 was present in 1 testcases

Requirement 6.1.2 was present in 1 testcases

Requirement 7.1 was present in 1 testcases

Requirement 7.2 was present in 1 testcases

Model Coverage Information

The testrun covered 4 of 10 requirements in the model

The testrun covered 1 of 3 use cases in the model

The testrun covered 8 of 24 transitions in the model

FIGURE 17.28
Example of statistical information from a test run.

MATERA also collects information from the UML models in order to calculate how different parts of the model have been covered.

Secondly, information concerning failed test cases is collected in order to identify from which parts of the system model a given failed test was generated. In this way, one can see which parts of the system model are not in sync with the real implementation, and therefore with the stake holder requirements. The previously mentioned test evaluation module also retrieves information about requirements from the EAST test logs and generates a set of OCL constraints used by MATERA. The generated constraints have two purposes.

- To locate in the model the uncovered requirements by the test run, in which case their parent requirement will also be located.
- To locate the models and their elements that "implement" a given uncovered or failed requirement.

Listing 17.12 shows an example of generated OCL constraints.

Listing 17.12
Example of generated OCL rules

```
-- Expressions for tracing uncovered or failed requirements in SysML
    requirements diagram:
not(Id = '3') and not(Id = '3.1') and not(Id = '3.1.1') and
not(Id = '6') and not(Id = '6.1') and not(Id = '6.1.1') and
not(Id = '7') and not(Id = '7.1') and not(Id = '7.2')

-- Expressions for finding uncovered or failed requirements placed on
    transitions:
not(clientDependency->exists(a |
(a.supplier.name->includes('3') or a.supplier.name->includes('3.1') or a.
    supplier.name->includes('3.1.1') or a.supplier.name->includes('6') or a.
    supplier.name->includes('6.1') or a.supplier.name->includes('6.1.1') or a
    .supplier.name->includes('7') or a.supplier.name->includes('7.1') or a.
    supplier.name->includes('7.2')))) and clientDependency->notEmpty())
```

The generated constraints are automatically loaded and interpreted in the MATERA framework, which will highlight the targeted elements. Once loaded and executed in MATERA, the first expression will trace and highlight in the diagram editor the set of requirements to which it is attached (see Figure 17.29). Similarly, the second rule will highlight the elements to which the previous requirements were linked. For instance, Figure 17.30 shows a screenshot of MagicDraw displaying the transitions in the state model to which *requirement 3.1.1* is linked.

17.8 Related Work

There is much literature on MBT in general and specific aspects in particular. For instance, Utting, Pretschner, and Legeard (2006) suggests a taxonomy for MBT, while in Hartman (2002) and Utting and Legeard (2006), the authors discuss MBT from a tools perspective, both academic and industrial. MBT techniques and tools have also been researched and developed in the context of the *Automated Generation and Execution of Test Suites for Distributed Component-based Software* (AGEDIS) (http://www.agedis.de) and *Deployment of Model-based Technologies to Industrial Testing* (D-MINT) (http://www.d-mint.org)

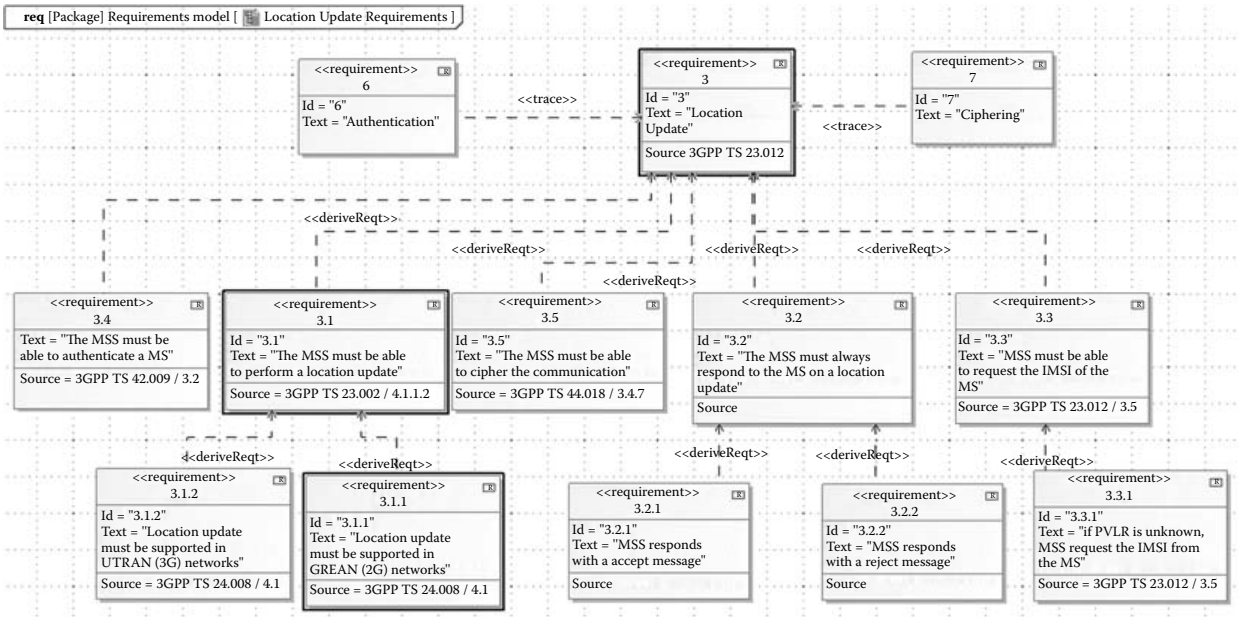


FIGURE 17.29
Back-tracing of requirements to the requirements model. (Reproduced from Abbors, F., Backlund, A., and Truscan, D., MATERA—An Integrated Framework for Model-Based Testing, *Proceedings: 2010 17th IEEE Conference and Workshops*, © 2010 IEEE.)

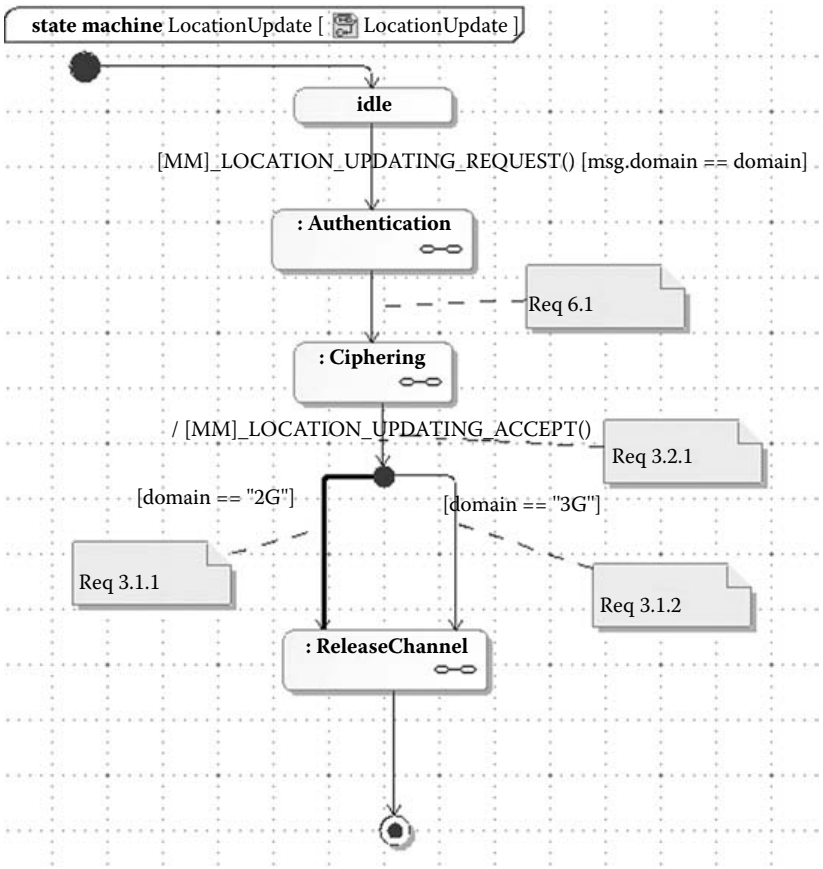


FIGURE 17.30
Back-tracing of requirements to a state machine model.

projects. The projects focused on the applicability of MBT techniques to industrial environments, and numerous publications and reports can be found on the web pages of each project. Both projects were applied to industrial case studies from different application domains, including telecommunications.

For instance, the authors describe in Born et al. (2004) a Model-driven Development and Testing process for Telecommunication Systems. A top-down approach based on CORBA is proposed in which the system is decomposed into increasingly smaller parts that can individually be implemented and tested. They make use of UML and profiles, in particular the Common Object Request Broker Architecture (CORBA) Component Model (CCM) (Object Management Group a), for specifying system models. These models are later on mapped onto software components using code generators, and executed within the CCM platform. Testing is also supported using specific UML profiles, in particular UML2 Testing Profile (U2TP), and by generators. The test design is tightly coupled with the system design so as to be able to reuse information provided in the system design, as soon as it becomes available. Testing is based on contracts built into different components and is derived from the models of the system. Test models are specified using U2TP and are used for the derivation of Testing and Test Control Notation (TTCN-3) tests. The TTCN-3 test are later automatically translated into executable Java byte code.

An approach similar to ours where several telecommunication case studies are evaluated against the Qtronic tool is discussed in Khan and Shang (2009). The main target of the work is to investigate the applicability of MBT in two telecommunication case studies. Another evaluation of MBT via a set of experiments run on Qtronic is described in Nordlund (2010). Both of those investigations draw the conclusion that MBT brings benefits in terms of automation and improved test coverage compared to traditional software testing.

Requirements traceability is a very popular topic in the software engineering and testing communities and has gained momentum in the domain of MBT in the context of automated test generation. However, as requirements change during the development life cycles of software systems, updating and managing traces have become tedious tasks. Researchers have addressed this problem by developing methods for automatic generations of traceability relations Hayes, Dekhtyar, and Osborne (2003), Spanoudakis et al. (2004), Cleland-Huang et al. (2005), Duan and Cleland-Huang (2006) by using information retrieval techniques to link artifacts together based on common keywords that occur in both the requirement description and in a set of searchable documents. Other approaches focus on annotating the model with requirements that are propagated through the test generation process in order to obtain a requirement traceability matrix Bouquet et al. (2005).

Work regarding requirements traceability similar to the MATERA approach is found in Bernard and Legeard (2007). There, the authors suggest an approach in which they annotate the UML system models with ad hoc requirement identifiers and use it to automatically generate a Traceability Matrix at the same time as the generated test cases. Their approach is embedded in the LEIROS Test Designer tool (SmartTesting). However, once the generated test cases are executed, requirements covered by tests are not traced back to the model.

17.9 Conclusions

Our experience indicates that MBT can already be used for productive testing in the telecommunication industry but there are some challenges. The challenges are related more to the tool integration than to MBT as a technology itself. If the challenges will be solved

properly, MBT will become even more effective technology within the telecommunication domain.

Initial expectations on MBT were that modeling is difficult and requires a lot of expertise and competence. Our experiences indicate the opposite. Modeling as a competence seems to be fairly easy to learn. At the same time, modeling helps gain and retain an overview of the system behavior much easier than using programming and scripting languages.

Automated test generation is clearly significantly different in comparison to manual test scripting. In test generation, the emphasis is on the design phase instead of the implementation of test cases. Based on our experience, the modifications on the model are propagated into the test cases significantly faster using test generation than manually modifying test scripts. By the same token, automated test generation forces exploiting a different strategy on tracing details, such as requirements, throughout testing compared to manual test scripting. Content of manually written test cases tends to be static and hence the names of the test cases are typically human readable names that testing staff learn fairly quickly. Because of these aspects, mapping of the test scripts and the requirements is fairly static. In the case of automatic test generation, the content of test cases is more dynamic. The content of the test cases may change significantly between test generation rounds and because of that, the names of the test cases do not reflect the content of the tests. Because of this, tracking the details of testing requires a new strategy. In the developed methodology, requirements are used to track test coverage, progress of testing, etc. aspects. Using requirements for tracking is natural because the requirements are present in product development. Hence, use of the requirements for tracking does not necessitate any additional and artificial details for the methodology.

During the development of the methodology, a lot of effort was put into tool integration. Despite the fact that the models were described using standardized modeling languages, such as UML and SysML, it was not possible to create a UML model using one tool and open it with another tool. Compared to text-based programming and testing languages (e.g., C++ and TTCN-3), this is a significant drawback for UML modeling. Model transformations between the tools can be used to circumnavigate this problem. However, seamless use of the tools in, for example, model validation, refactoring, and traceability analysis, requires frequent interaction between the tools. Consequently, transformations will be performed frequently, and hence there would be a tool chain-induced performance penalty. In addition, implementing a number of model transformations increases the costs of the tool chain.

References

Abbors, F. (2009a). An Approach for Tracing Functional Requirements in Model-Based Testing. Master's thesis, Åbo Akademi University.

Abbors, F., Bäcklund, A., and Truscan, D. (2010). MATERA—An integrated framework for model-based testing. In *17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2010)*, Pages: 321–328. IEEE Computer Society's Conference Publishing Services (CPS), Los Alamitos, CA.

Abbors, F., Pääjärvi, T., Teittinen, R., Truscan, D., and Lilius, J. (2009). Transformational support for model-based testing—From UML to QML. In *Proceedings of 2nd Workshop on Model-based Testing in Practice (MOTIP'09)*, Enschede, the Netherlands.

Abbors, J. (2009b). Increasing the quality of UML Models Used for Automated Test Generation. Master's thesis, Åbo Akademi University.

Beizer, B. (1990). *Software Testing Techniques* (2nd ed.), Van Nostrand Reinhold Co., New York.

Bérnard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., and McKenzie, P. (2001). *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer-Verlag New York, Inc. New York.

Bernard, E. and Legeard, B. (2007). Requirements traceability in the model-based testing process. *Software Engineering, ser. Lecture Notes in Informatics*, Pages: 45–54.

Born, M., Schieferdecker, I., Gross, H., and Santos, P. (2004). Model-driven development and testing—A case study. In *1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application, number TR-CTIT-04-12 in CTIT Technical Report*, Pages: 97–104. University of Twente, Enschede, the Netherlands.

Bouquet, F., Jaffuel, E., Legeard, B., Peureux, F., and Utting, M. (2005). Requirements traceability in automated test generation: Application to smart card software validation. In *Proceedings of the 1st International Workshop on Advances in Model-Based Testing*, Pages: 1–7. ACM, New York, NY.

Cleland-Huang, J., Settimi, R., Duan, C., and Zou, X. (2005). Utilizing supporting evidence to improve dynamic requirements traceability. In *13th IEEE International Conference on Requirements Engineering, 2005. Proceedings*, Pages: 135–144.

Conformiq (2009a). Conformiq Qtronic. Web page. http://www.conformiq.com/products.php. Accessed on June 13, 2011.

Conformiq (2009b). Conformiq Qtronic User Manual. http://www.conformiq.com/downloads/Qtronic2xManual.pdf. Accessed on June 13, 2011.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. (1999). Model-based testing in practice. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, Pages: 285–294. IEEE Computer Society Press, Los Alamitos, CA.

Duan, C. and Cleland-Huang, J. (2006). Visualization and analysis in automated trace retrieval. In *Requirements Engineering Visualization, 2006. REV'06. First International Workshop on*, Pages: 5–5.

Hartman, A. (2002). Model based test generation tools. *Agedis Consortium, URL: http://www. agedis. de/documents/ModelBasedTestGenerationTools_cs. pdf*. Accessed on June 13, 2011.

Hayes, J., Dekhtyar, A., and Osborne, J. (2003). Improving requirements tracing via information retrieval. In *11th IEEE International Requirements Engineering Conference, 2003. Proceedings*, Pages: 138–147.

Kaaranen, H., Ahtiainen, A., Laitinen, L., Naghian, S., and Niemi, V. (2005). *UMTS Networks, 2nd Edition*. John Wiley & Sons, Ltd, Chichester, UK.

Khan, M. and Shang, S. (2009). Evaluation of Model Based Testing and Conformiq Qtronic. Master's thesis, Linköping University. http://liu.diva-portal.org/smash/record.jsf?pid=diva2:249286.

No Magic (2009). MagicDraw. Web page. http://www.magicdraw.com/. Accessed on June 13, 2011.

Malik, Q. A., Jääskeläinen, A., Virtanen, H., Katara, M., Abbors, F., Truscan, D., and Lilius, J. (2009). Using system models vs. test models in model-based testing. In *Proceedings of "Opiskelijoiden minikonferenssi" at "Tietotekninen tuki ohjelmoinnin opetuksessa."*

Nethawk (2009). Nethawk EAST simulator. Web page. https://www.nethawk.fi/products/nethawk_simulators/. Accessed on June 13, 2011.

Nokia Siemens Networks (July 2009). Nokia Siemens Networks Company Profile. Web site. http://www.nokiasiemensnetworks.com/about-us/company. Accessed on June 13, 2011.

Nordlund, J. (2010). Model-Based Testing: An Evaluation. Master's thesis, Karlstad University. http://kau.diva-portal.org/smash/record.jsf?pid=diva2:291718. Accessed on June 13, 2011.

Object Management Group. CORBA Component Model. http://www.omg.org/spec/CCM/4.0/. Accessed on June 13, 2011.

Object Management Group. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/2.0/. Accessed on June 13, 2011.

Object Management Group. Systems Modeling Language. http://www.omg.org/spec/SysML/1.1/. Accessed on June 13, 2011.

Object Management Group. Unified Modeling Language (UML). http://www.omg.org/spec/UML/2.1.2/. Accessed on June 13, 2011.

Prenninger, W., El-Ramly, M., and Horstmann, M. (2005). *Model-Based Testing of Reactive Systems*, Chapter Case Studies. Number 3472 in Advance Lectures of Computer Science. Springer, New York.

SmartTesting. CertifyIt, http://smartesting.com/index.php/cms/en/product/certify-it. Accessed on June 13, 2011.

Spanoudakis, G., Zisman, A., Perez-Minana, E., and Krause, P. (2004). Rule-based generation of requirements traceability relations. *The Journal of Systems & Software*, 72(2): 105–127.

The 3rd Generation Partnership Project (2005). 3GPP TS 23.002 Network Architecture, v. 6.10.0. Web site. http://www.3gpp.org/ftp/Specs/html-info/23002.htm. Accessed on June 13, 2011.

Utting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Utting, M., Pretschner, A., and Legeard, B. (2006). A taxonomy of model-based testing. Working Paper Series, ISSN 1170-478X. http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf. Accessed on June 13, 2011.

WireShark. WireShark Network Protocol Analyzer. http://www.wireshark.org/. Accessed on June 13, 2011.

Paper V

Model-based Testing of Web Services Using Probabilistic Timed Automata

Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres

Originally published *2013 Proceeding of 9th International Conference on Web information Systems and Technologies*. SCITEPRESS. May 2013, Aachen, Germany.

Model-Based Performance Testing of Web Services Using Probabilistic Timed Automata

Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres
Department of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5 A, Turku, Finland
{*fabbors, tahmad, dtruscan, iporres*}*@abo.fi*

Keywords: Performance testing, Performance monitoring, Load Generation, Probabilistic Timed Automata

Abstract: In this paper, we present an approach for performance testing of web services in which we use abstract models, specified using Probabilistic Timed Automata, to describe how users interact with the system. The models are used in the load generation process to generate load against the system. The abstract actions from the model are sent in real-time to the system via an adapter. Different performance indicators are monitored during the test session and reported at the end of the process. We exemplify with an auction web service case study on which we have run several experiments.

1 Introduction

Today, we see advancements in cloud computing and more and more software applications being adapted to a cloud environment. Applications deployed in the cloud are delivered to users as a service, without the need for the users to install anything. This means that most of the processing is done on the server side and this puts a frightful amount of stress on the back-end of the system. Performance characteristics such as throughput, response times, and resource utilization are crucial quality attributes of such applications and systems.

The purpose of performance testing is to determine how well the system performs in terms of responsiveness, stability, and resource utilization under a particular synthetic workload in a controlled environment. The synthetic workload (Ferrari, 1984) should mimic the expected workload (Shaw, 2000) as closely as possible, once the system is in operational use, otherwise it is not possible to draw any reliable conclusions from the test results.

Performance tests are typically implemented as usage scenarios that are either manually scripted (e.g., using *httperf* or *JMeter*) or pre-recorded (e.g., using Selenium (SeleniumHQ, 2012) in the case of web applications). The usage scenarios are then executed concurrently against the system under test. A major drawback with this approach is that the manually coded scripts and pre-recorded scenarios seldom represent real-life traffic and that certain combinations of user inputs may remain untested. Repeating the same script over and over may lead to unrealistic results because of caching and other operating system optimization mechanisms. Performance testing is done efficiently when it is executed in an iterative manner and uses techniques that simulate real life work load as closely as possible (Menasce, 2002). This means that load is incrementally increased until a certain threshold (saturation) is reached, beyond which the performance of the system begins to degrade.

In this paper, we propose a model-based approach to evaluate the performance of a system by incrementally exercising different kinds of loads on the system. The main contributions of this work are:

- we use abstract models, specified as Probabilistic Timed Automata (PTA) to model the user profiles, including the *actions* or *sequences of actions* the user can send, the probabilistic distribution of the actions, and individual *think time* for each action,
- the load is generated in real-time from these models and sent to the system under test (SUT) via an adapter which converts abstract actions into concrete interactions with the SUT and manages data dependencies between different actions;

The rest of the paper is structured as follows: In Section 2 we give an overview of the work related to our approach. Section 3 presents our model-based testing process, while Section 4 presents an auction web service case study and an experiment using our approach. Finally, in Section 5, we present our conclusions and we discuss future work.

2 Related Work

There is already a large body of work on workload characterization and a more limited one on load generation from performance models. In the following, we briefly enumerate several works that are closer to our approach.

Barna et al., (Barna et al., 2011) present a model-based testing approach to test the performance of a transactional system. The authors make use of an iterative approach to find the workload stress vectors of a system. An adaptive framework will then drive the system along these stress vectors until a performance stress goal is reached. They use a system model, represented as a two-layered queuing network, and they use analytical techniques to find a workload mix that will saturate a specific system resource. Their approach differs from ours in the sense that they use a model of the system instead of testing against a real implementation of a system.

Other related approaches can be found in (Shams et al., 2006) and (Ruffo et al., 2004). In the former, the authors have focused on generating valid traces or a synthetic workload for inter-dependent requests typically found in sessions when using web applications. They describe an application model that captures the dependencies for such systems by using Extended Finite State Machines (EFSMs). Combined with a workload model that describes session inter-arrival rates and parameter distributions, their tool *SWAT* outputs valid session traces that are executed using a modified version of *httperf* (Mosberger and Jin, 1998). The main use of the tool is to perform a sensitivity analysis on the system when different parameters in the workload are changed, e.g., session length, distribution, think time, etc. In the latter, the authors suggest a tool that generates representative user behavior traces from a set of Customer Behavior Model Graphs (CBMG). The CBMG are obtained from execution logs of the system and they use a modified version of the *httperf* utility to generate the traffic from their traces. The methods differ from our approach in the sense they both focus on the trace generation and let other tools take care of generating the load/traffic for the system, while we do on-the-fly load generation from our models.

(Denaro et al., 2004) propose an approach for early performance testing of distributed software when the software is built using middleware components technologies, such as J2EE or CORBA. Most of the overall performance of such a system is determined by the use and configuration of the middleware (e.g. databases). They also note that the coupling between the middleware and the application architecture determines the actual performance. Based on architectural designs of an application the authors can derive application-specific performance tests that can be executed on the early available middleware platform that is used to build the application with. This approach differs from ours in that the authors mainly target distributed systems and testing of the performance of middleware components.

3 Performance Testing Process

Our performance testing process is depicted in Figure 1. In brief, we build a workload model of the system by analyzing different sources of information, and subsequently we generate load in on-the-fly against the system. During the process, different performance indicators are measured and a test report is created at the end.

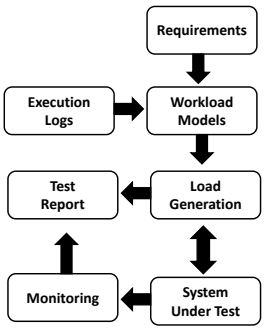


Figure 1: Our performance testing process.

In our work, we have used various *Key Performance Indicators* (KPIs) to provide quantifiable measurements for our performance goals. For instance, we specify the *target KPIs* before the testing procedure is started and later on we compare them against the actual *measured KPIs*.

3.1 Workload characterization

The first step in our process is characterizing the workload of the system. According to (Menasce and Almeida, 2001), the workload of a system can be defined as the set of all inputs the system receives from the environment during any given period of time.

Traditionally, performance analysis starts first with identifying key performance scenarios, based on the idea that certain scenarios are more frequent than others or certain scenarios impact more on the performance of the system than other scenarios. A performance scenario is a sequence of actions performed by an identified group of users (Petriu and Shen, 2002).

In order to build the workload model, we start by looking and analyzing the requirements and the system specifications, respectively. During this phase we try to form an understanding of how the system is

used, what are the different types of users, and what are the key performance scenarios that will impact most on the performance of the system. A user type is characterized by the distribution and the types of actions if performs.

The main sources of information for workload characterization are: Service Level Agreements (SLAs), system specifications, and standards.

By using these sources we identify the inputs of the system with respect to types of transactions (actions), transferred files, file sizes, arrival rates, etc. following the generic guidelines discussed in (Calzarossa et al., 2000). In addition, we extract information regarding the KPI's, such as the number of concurrent users the system should support, expected throughput, response times, expected resource utilization demands etc. for different actions under a given load. We would like to point out that this is a manual step in the process. However, automating this step could be achieved analyzing log files of the system and using various clustering algorithms for determining e.g., different user types, which is subject for future work.

The following steps are used for analyzing the workload:

1. Identify the actions that can be executed against the system.
   (a) Determine the required input data for each action. For instance, the request type and the parameters.
   (b) Identify dependencies between actions. For example, a user can not execute a logout action before a login action.
2. Identify the most relevant user types, based for instance on the amount of interactions with the system.
3. Define the distribution of actions that are performed by each user type.
4. Estimate an average *think time* per action.

With *think time* we refer to the time between two consecutive actions. In our approach, the think time for the same action can vary from one user to another, or from one test scenario to another.

3.2 Workload models

The results of the workload characterization are aggregated in a workload model based on Probabilistic Timed Automata.

We take the definition of a *probabilistic timed automaton* (PTA) as defined by (Kwiatkowska et al., 2006). A (PTA) P = (L,$\bar{l}$, X, $\sum$, *inv*, *prob*) is a tuple consisting of a finite set L of locations with the initial location $\bar{l} \in$ L; a finite set X of clocks; a finite set of $\sum$ of actions; a function *inv* : L $\rightarrow$ CC(X) associating an invariant condition with each location, where CC(X) is a set of clock constraints over X; a finite set *prob* $\subseteq$ L $\times$ CC(X) $\times$ $\sum$ $\times$ Dist(2^X $\times$ L) of probabilistic transitions, such that, for each l $\in$ L, there exists at least one (l,_,_,_) $\in$ *prob*; and a labeling function δ : L $\rightarrow$ 2^{AP}, where AP denote a set of atomic propositions.

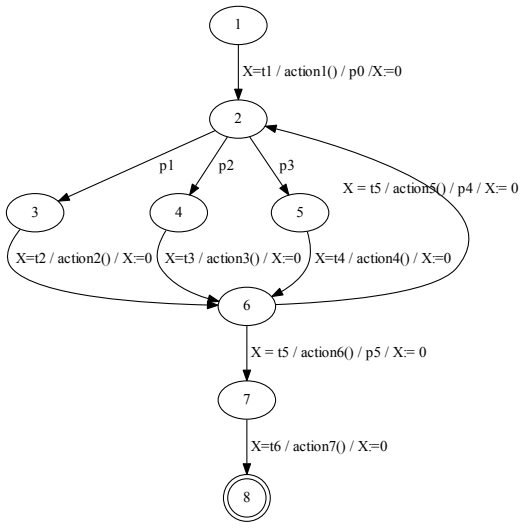


Figure 2: Example of a probabilistic timed automaton.

A probabilistic transition *(l,g,p,a)* $\in$ *prob* is a quadruple containing (1) a source location *l*, (2) a clock constraint *g*, called *guard* or *invariant condition*, (3) a *probability p*, and (4) an *action*. The *probability* indicates the chance of that transition being taken. The *action* describes what action to take when the transition is used, and the *clock* indicates how long to wait before firing the transition. The behavior of a PTA is similar to that of a timed automaton (Alur and Dill, 1994): in any location, time can advance as long as the invariant holds, and a probabilistic transition can be taken if its guard is satisfied by the current values of the clocks. Every automaton has an *end location*, depicted with a double circle, which will eventually be reached. It is possible to specify loops in the automaton. We note that not all transitions have both a *guard* and a *probability*. For simplicity, we do not explicitly specify location invariants, but they implicitly evaluate to **true**. One such workload model is created for each identified user type.

3.3 Load generation

The resulting workload models are used for generating load in real-time against the system under test, by

creating traces from the corresponding PTA. The user types are selected based on their reciprocal distribution. The PTA of each user type will be executed concurrently by selecting the corresponding actions and sending them to the system. By executing the PTA of a given user, in each step an action is chosen based on the probabilistic values in the automaton.

The load generation is based on a deterministic choice with a probabilistic policy. This introduces certain randomness into the test process and that can be useful for uncovering certain sequences of actions which may have a negative impact of the performance. Such sequences would be difficult or maybe impossible to discover if static test scripts are used, where a fixed order of the actions is specified, and repeated over and over again. Every PTA has an *exit* location which will eventually be reached. By modifying the probability for the *exit* action, it is also possible to adjust the average length of the generated sequences.

3.4 Performance monitoring

During the load generation, we constantly monitor target KPIs for the entire test duration. At the end, we collect all the gathered data and compute descriptive statistics, like the mean and peak response times for different actions, number of concurrent users, the amount of transferred data, the error rate, etc. All the gathered information is presented in a test report. The resource utilization of the system under test is also monitored and reported. Besides computing different kinds of statistical values from the raw data we have, the test report also contains graphs such as how the response time varied over time with the number of concurrent users. The test report also shows the CPU, disk, network and memory usage on the target system.

Tool support for load generation is provided via the MBPeT tool (Abbors et al., 2012). Due to space limitations we defer more details about the approach and support to (Ahmad et al., 2013)

4 Case Study and Experiments

In this section, we demonstrate our approach by using it to evaluate the performance of an *auction web service*, generically called YAAS. The YAAS application was developed as a stand-alone application and is used for the evaluation of our approach. The YAAS has a *RESTful* (Richardson and Ruby, 2007) interface based on the HTTP protocol and allows registered users to create, change, search, browse, and bid on auctions that other users have created. The application maintains a database of the created auctions and the bids that other users have placed on the auctioned objects. The YAAS application is implemented using Python (Python, 2012) and the Django (Django, 2012) framework.

Test Architecture. The test architecture is shown in Figure 4. The MBPeT tool has a scalable architecture where a master node controls several slave nodes. The SUT runs an instance of the YAAS application on top of the Apache web server. All nodes (master, slaves, and the server) feature an 8-core CPU, 16 GB of memory, 1Gb Ethernet, 7200 rpm hard drive, and Fedora 16 operating system. The nodes were connected via a 1Gb Ethernet.

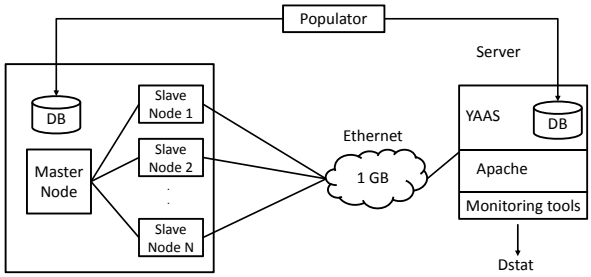


Figure 4: A caption of the test architecture.

A *populator* script is used to generate input data (i.e., populate the test databases) on both the client and server side, before each test session. This ensures that the test data on either side is consistent and easy to rebuild after each test session.

Workload Modeling. We analyzed the workload following the steps described in Section 3. Based on this analysis, three user types were identified: *aggressive_bidders*, *passive_bidders*, and *non-bidders*. From this information, we constructed a PTA model for each user type. Figure 3 shows the PTA for a *aggressive_bidder*.

Figure 3 shows that each action has a think time parameter, modeled as a clock variable associated with it, that specifies how much time should elapse before firing a transition. This variable is denoted with the symbol X and it is reset to 0 after the transition is fired. Upon firing the transition, the action associated with that transition is sent to the SUT.

Adapter. An adapter is used to translate abstract actions generated from the model into concrete HTTP requests by adding the necessary HTTP parameters and encapsulation to the SUT. All slaves run identical adapters. The models as such are system independent, but an adapter module need to be written for every system that one chooses to interface with. Since YAAS is based on the HTTP protocol, it will understand the basic HTTP commands like GET, POST, PUT, etc. Whenever a new action is selected from

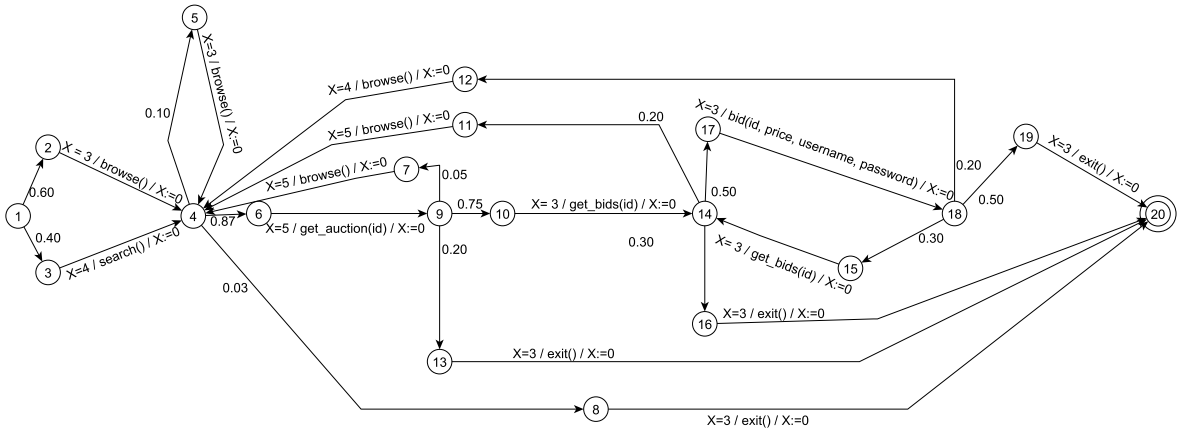


Figure 3: PTA model for an *aggressive-bidder* user type.

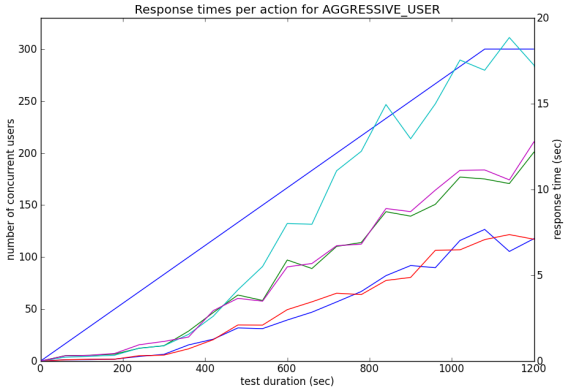


Figure 5: Average response times for *get_auction* and *get_bids* (bottom), *search* and *browse* (middle), and *bid* (top) when ramping up from 0 to 300 users.

the model, the corresponding HTTP request is created and, when needed, the associated data is automatically attached to the request from the local database.

Experiments. In the case study, an experiment was conducted to find out how the YAAS application performs under load. As a rule of thumb for ensuring accurate results, the experiment was run three times.

In the experiment, we set out to test *how many concurrent users the host node can support without exceeding the specified target response time values*.

Table 1 shows the average and max response time limits (see column *Target Response Time*) that were selected for each type of action. For instance, the average response time limit for action *browse()* was set to 4.0 seconds, while the max response time was set to 8.0 seconds. If any of the set limits (average and max) are breached during the test run, the tool will mark the time of the breach and the number of concurrent users at that time (see Table 1 - *Time of breach*). The length of the test run was 20 minutes (1200 seconds). Figure 5 shows how the response times of different actions increase over time for the *aggressive_user* type as the number of concurrent users are ramped up from 0 to 300. In this experiment the tool generated a total of 1504 unique test sequences form the models. Several of the unique test sequences were executed more than 100 times and the variance on the test sequence length was from 1 up to 50 actions.

Table 1 also shows the time when a target response time (average and/or max) value was exceeded and the number of concurrent users at that time. For example, the average response time for the *search()* action was exceeded at 229 seconds into the test run by the *aggressive user* type. The tool was when running 64 concurrent users. Form this table we concluded that the current configuration of the server can support a maximum of 64 concurrent users before exceeding the threshold value of 3 seconds set for action *search()*. A closer inspection of the monitored values of the server showed that the database was the bottleneck, due to the fact a *sqlite database* was used and the application locked the whole database for write operations.

Additional experiments, including a comparison of our approach against JMeter can be found in (Ahmad et al., 2013). The experiment showed that out tool has similar capabilities as JMeter for instance when comparing the throughput (actions/sec) against the SUT.

5 Conclusion and Future Work

In this paper, we have presented a model-based performance testing approach that uses probabilistic models to generate synthetic load in real-time. The

	Target Response Time		Non-Bidders (22 %)		Passive Users (33 %)		Aggressive users 45 %		Verdict
Actions	Average (sec)	Max (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Time of breach (sec)	Pass/fail
browse()	4.0	8.0	279 (78 users)	394 (110 users)	323 (90 users)	394 (110 users)	279 (78 users)	394 (110 users)	Failed
search(string)	3.0	6.0	279 (78 users)	394 (110 users)	279 (78 users)	394 (110 users)	229 (64 users)	327 (92 users)	Failed
get_action(id)	2.0	4.0	280 (79 users)	325 (91 users)	279 (78 users)	279 (78 users)	276 (77 users)	325 (91 users)	Failed
get_bids(id)	3.0	6.0	279 (78 users)	446 (130 users)	325 (91 users)	394 (110 users)	327 (92 users)	394 (110 users)	Failed
bid(id,price, username, password)	5.0	10.0	—-	—-	327 (92 users)	474 (132 users)	328 (92 users)	468 (131 users)	Failed

Table 1: Response time measurements for user actions when ramping up from 0 to 300 users.

models are based on the Probabilistic Timed Automata, and include statistical information that describes the distribution between different actions and corresponding think times. With the help of probability values, we can make it so that a certain action is more likely to be chosen over another action, whenever the virtual user encounters a choice in the PTA. We believe that the PTA models are well suited for performance testing and that the probability aspect that the PTA holds is good for describing dynamic user behavior, allowing us to include a certain level of randomness in the load generation process. This is important because we wanted the virtual users to be able to mimic real user behavior as closely as possible, and minimize the effect of caches on the performance evaluation.

The approach is supported by a set of tools, including the MBPeT load generator. MBPeT has a scalable distributed architecture which can be easily deployed to cloud environments. The tool has a ramping feature which describes at what rate new users are added to the system and also supports the ability to specify a think time. When the test duration has ended the MBPeT tool will gather measured data, process it and create a test report.

In the future we will look into if parts of the model creation can be automated. At the moment it is done manually. There are indications that certain parts of creating the models can be automated e.g. by automatically analyzing the log data and using different clustering algorithms.

REFERENCES

Abbors, F., Ahmad, T., Truscan, D., and Porres, I. (2012). MBPeT: A Model-Based Performance Testing Tool. *2012 Fourth International Conference on Advances in System Testing and Validation Lifecycle*.

Ahmad, T., Abbors, F., Truscan, D., and Porres, I. (2013). Model-Based Performance Testing Using the MBPeT Tool. Technical Report 1066, Turku Centre for Computer Science (TUCS).

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235.

Barna, C., Litoiu, M., and Ghanbari, H. (2011). Model-based performance testing (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 872–875, New York, NY, USA. ACM.

Calzarossa, M., Massari, L., and Tessera, D. (2000). Workload Characterization Issues and Methodologies. In *Performance Evaluation: Origins and Directions*, pages 459–481, London, UK, UK. Springer-Verlag.

Denaro, G., Polini, A., and Emmerich, W. (2004). Early performance testing of distributed software applications. In *Proceedings of the 4th international workshop on Software and performance*, WOSP '04, pages 94–103, New York, NY, USA. ACM.

Django (2012). Online at https://www.djangoproject.com/.

Ferrari, D. (1984). On the foundations of artificial workload design. In *Proceedings of the 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '84, pages 8–14, New York, NY, USA. ACM.

Kwiatkowska, M., Norman, G., Parker, D., and Sproston, J. (2006). Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29:33–78.

Menasce, D. A. (2002). Load Testing of Web Sites. *IEEE Internet Computing*, 6:70–74.

Menasce, D. A. and Almeida, V. (2001). *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Mosberger, D. and Jin, T. (1998). httperfa tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37.

Petriu, D. C. and Shen, H. (2002). Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. pages 159–177. Springer-Verlag.

Python (2012). Python programming language. Online at http://www.python.org/.

Richardson, L. and Ruby, S. (2007). *Restful web services*. O'Reilly, first edition.

Ruffo, G., Schifanella, R., Sereno, M., and Politi, R. (2004). WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance. *Network Computing and Applications, IEEE International Symposium on*, 0:77–86.

SeleniumHQ (2012). Online at http://seleniumhq.org/.

Shams, M., Krishnamurthy, D., and Far, B. (2006). A model-based approach for testing the performance of web applications. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, pages 54–61, New York, NY, USA. ACM.

Shaw, J. (2000). Web Application Performance Testing – a Case Study of an On-line Learning Application. *BT Technology Journal*, 18(2):79–86.

Paper VI

Performance Testing in the Cloud using MBPeT

Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres

Originally published *2013 Developing Cloud Software*. Turku Centre for Computer Science (TUCS), 2013.

6 Performance Testing in the Cloud Using MBPeT

Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres
Department of Information Technologies
Åbo Akademi University, Turku, Finland
Email: {fredrik.abbors, tanwir.ahmed, dragos.truscan, ivan.porres}@abo.fi

Abstract–We present a model-based performance testing approach using the MBPeT tool. We use of probabilistic timed automata to model the user profiles and to generate synthetic workload. The MBPeT generates the load in a distributed fashion and applies it in real-time to the system under test, while measuring several key performance indicators, such as response time, throughput, error rate, etc. At the end of the test session, a detailed test report is provided. MBPeT has a distributed architecture and supports load generation distributed over multiple machines. New generation nodes are allocated dynamically during load generation. In this book chapter, we will present the MBPeT tool, its architecture, and demonstrate its applicability with a set of experiments on a case study. We also show that using abstract models for describing the user profiles allows us quickly experiment different load mixes and detect worst case scenarios.

Keywords-Performance testing, model-based testing, MBPeT, cloud.

6.1 Introduction

Software testing is the process of identifying incorrect behavior of a system, also known as revealing defects. Uncovering these defects, typically, consists of running a batch of software tests (*test suite*) against the software itself. In some sense, a second software artefact is built to test the primary one. This is normally referred to as *functional testing*. A software test compares the actual output of the system with the expected output for a particular

known input. If the actual output is the same as the expected output the test passes, otherwise a test fails and a defect is found. Software testing is also the means to assess the quality of a software product. The fewer the defects found during testing, the higher the quality is of that software product. However, not all software defects are related to functionality. Some systems may stop functioning or may prevent other users to access the system simple because the system is under a heavy workload with which it cannot cope. Performance testing is the means of detecting such errors.

Performance testing is the process of determining how a software system performs in terms of responsiveness and stability under a particular workload. The purpose of the workload is that it should match the expected workload (the load that normal users put on the system when using it) as closely as possible. This can be achieved by running a series of tests in parallel, but instead of focusing on the right output the focus is shifted towards measuring non-functional aspects, i.e. the time between input and output (response time) or number of requests processed in a second (throughput).

Traditionally, performance testing has been conducted by running a number of predefined scenarios (or scripts) in parallel. One drawback to this approach is that real users do not behave as static scripts. This can also lead to certain paths in the system being left untested or that certain caching mechanisms in the system kick in due the repetitiveness of the test scripts.

Software testing can be extremely time consuming and costly. In 2005, Caper Jones - chief scientist of Software Productivity Research in Massachusetts - estimated that as much as 60 percent of the software work in the United States was related to detecting and fixing defects [1]. Another drawback is that software testing, as well as performance testing, involves tedious manual work when creating test cases. A software system typically undergoes a lot of changes during its lifetime. Whenever a piece of code is changed, a test has to be updated or created to show that the change did not break any existing functionality or introduce any new defects. This adds more time and cost to testing. In the case of performance testing this implies that one has to be able to benchmark quickly and effectively to check if the performance of the system is affected by the change of the code.

Research effort have be put into solving this dilemma. One of the most promising techniques is Model-Based Testing (MBT). In MBT, the central artefact is a system model. The idea is that the model represents the behavior or the use of the system. Tests are then automatically generated form the model. In MBT the focus has shifted from manually creating tests to maintaining a model that represents the behavior of the system. Due to the fact that tests are automatically generated from a model, MBT copes better with changing requirements and code than traditional testing. Research has

shown that MBT could reduce the total testing costs with 15 percent [8]. MBT has mostly been targeted towards functional testing, however, there exist a few tools that utilizes the power of MBT in the domain of performance testing. In our research we make use of the advantages of MBT in our performance testing approach.

MBPeT is a Python-based tool for performance testing. Load is generated from *probabilistic timed automata* (PTA) models describing the behavior of groups of virtual users. The models are then executed in parallel to get a semi-random workload mix. The abstract PTA models are easy to create and update, facilitating quick iteration cycles. During the load generation phase, the tool also monitors different *key performance indicators* (KPIs) such as response times, throughput, memory, CPU, disk, etc. The MBPeT tool has a distributed architecture where one master node controls several slave node or load generator. This facilitates deployment to a cloud environment. Besides monitoring, the tool also produces a performance test report at the end of the test. The report contains information about the monitored KPIs, such as response times, throughput etc, but also graphs showing how CPU, memory, disk, network utilization varied during a performance test session.

The rest of the report is structured as follows: we briefly enumerate several related works in the following section. Then, is Section 6.3, we briefly describe the load generation process. In Section 6.4, we give an overview of the architecture of the tool. In Section 6.5, we describe how the workload models are created and discuss the probabilistic timed automata formalism. In Section 6.6, we discuss the performance testing process in more detail. In Section 6.7, we present a auction web service case study and a series of experiments using our tool. Finally, in Section 6.8 we present our conclusions and discuss future work.

6.2 Related Work

There exist a plethora of commercial performance testing tools. In the following, we briefly enumerate couple of popular performance testing tools. FABAN is an open source framework for developing and running multi-tier server benchmarks [18]. FABAN has a distributed architecture meaning load can be generated from multiple machines. The tool has three main components: *A harness* - for automating the process of a benchmark run and providing a container for the benchmark driver code, a *Driver framework* - provides an API for people to develop load drivers, and an *Analysis tool* - to provide comprehensive analysis of the data gathers for a test. Load is generated by running multiple scripts in parallel. JMeter [19] is an open source

Java tool for load testing and measuring performance, with the focus on web applications. Jmeter can be set up in a distributed fashion and load is generated from manually created scenarios that are run in parallel. Httperf [6] is a tool for measuring the performance of web servers. Its aim is to facilitate the construction of both micro and macro-level benchmarks. Httperf can be set up to run on multiple machines and load is generated from pre-defined scripts. LoadRunner [7] is a performance testing tool from Hewlett-Packard for examining system behavior and performance. The tool can be run in a distributed fashion and load is generated from pre-recorded scenarios.

Recently several authors have focused on using models for performance analysis and estimation, as well as for load generation. Barna et al., [2] present a model-based testing approach to test the performance of a transactional system. The authors make use of an iterative approach to find the workload stress vectors of a system. An adaptive framework will then drive the system along these stress vectors until a performance stress goal is reached. They use a system model, represented as a two-layered queuing network, and they use analytical techniques to find a workload mix that will saturate a specific system resource. Their approach differs from ours in the sense that they use a model of the system instead of testing against a real implementation of a system.

Other related approaches can be found in [16] and [15]. In the former, the authors have focused on generating valid traces or a synthetic workload for inter-dependent requests typically found in sessions when using web applications. They describe an application model that captures the dependencies for such systems by using Extended Finite State Machines (EFSMs). Combined with a workload model that describes session inter-arrival rates and parameter distributions, their tool *SWAT* outputs valid session traces that are executed using a modified version of *httperf* [12]. The main use of the tool is to perform a sensitivity analysis on the system when different parameters in the workload are changed, e.g., session length, distribution, think time, etc. In the latter, the authors suggest a tool that generates representative user behavior traces from a set of Customer Behavior Model Graphs (CBMG). The CBMG are obtained from execution logs of the system and they use a modified version of the *httperf* utility to generate the traffic from their traces. The methods differ from our approach in the sense they both focus on the trace generation and let other tools take care of generating the load/traffic for the system, while we do on-the-fly load generation from our models.

Denaro [4] proposes an approach for early performance testing of distributed software when the software is built using middleware components technologies, such as J2EE or CORBA. Most of the overall performance of

such a system is determined by the use and configuration of the middleware (e.g. databases). They also note that the coupling between the middleware and the application architecture determines the actual performance. Based on architectural designs of an application the authors can derive application-specific performance tests that can be executed on the early available middleware platform that is used to build the application with. This approach differs from ours in that the authors mainly target distributed systems and testing of the performance of middleware components.

6.3 The Performance Testing Process

In this section we are briefly going to describe the steps of the performance testing process. A more detailed description is given in Section 6.6.

6.3.1 Model Creation

Before we start generation load for the system we first have to create a load profile or a load model that describe the behavior of the users. Since we can not have a model for each individual user we have to create one or several models that represent the behavior for a larger group of users. These models describe how a groups of virtual users (VUs) behave and they are simplified models of how a real users would behave. Section 6.5 gives more details of how the models are constructed. Essentially, we use probabilistic timed automata (PTA) to specify user behavior which describe in an abstract way the sequence of actions a VU can execute against the system and their probabilistic distribution.

6.3.2 Model Validation

Once the models have been created they are checked for consistency and correctness. For instance, we check that the models have a start and end point, that there are no syntactical errors in the models, and that the probabilities and actions have been defined correctly. Once the models have been checked by the MBPeT tool we start generating load for the system under test (SUT).

6.3.3 Test Setup

Before we can actually start generating load we need to set up everything correctly so that the MBPeT can connect to the SUT and generate the appropriate amount of load. To do that one have to fill in a settings file. This file contains e.g., the IP-address of the SUT, what load models to use, how

many parallel virtual users to simulate, ramp up period, and the duration of the performance test. The MBPeT tool needs this information in order to be able to generate the right amount of load.

The tester also needs to implement an adapter for the tool. Every SUT will have its own adapter implementation. The purpose of the adapter is to translate the abstract actions found in the model into concrete actions understandable by the SUT. In case of a web page, a *browse* action would need to be translated into a HTTP *GET* request.

6.3.4 Load Generation

Once everything is set up, load generation begins. The MBPeT tool generates load from the models by starting a new process for every simulated user. Inside that process load is generated by executing the PTA model. For more details please see Section 6.6.2. Please see Section 6.5.2 for more information on PTAs.

6.3.5 Monitoring

During the load testing phase the MBPeT tool monitors the traffic sent on the network to the SUT. The tool monitors the throughput and response time for every action sent to the system. If there is a possibility to connect to the SUT remotely, the MBPeT tool can also monitor the utilization of the CPU, memory, network, disk, etc. This information can be very useful when trying to identify potential bottlenecks in the system. Once the test run is complete and all information is gathered, the tool will create a test report.

6.3.6 Test Reporting

The test report contains information about the parameters monitored during the performance test. It gives statistical values of the mean and max response time for individual actions and displays graphs that show how the repones time varied over time when the load increases. If the tool can be connected remotely to the SUT, the test report will also show how the CPU, memory, and disk was utilized over time when the load was applied to the SUT. Both of these sources of information can be helpful when trying to pin the a potential bottleneck in the system.

6.4 MBPeT Tool Architecture

MBPeT has a distributed architecture. It consists of two types of nodes: a master node and slave nodes. A single master node is responsible of initiating and controlling multiple remote slave nodes, as shown in Figure 6.1. Slave nodes are designed to be identical and generic, in a sense that they do not have prior knowledge of the SUT, its interfaces, or the workload models. That is why for each test session, the master gathers and parses all the required information regarding the SUT and the configuration for each test session and sends that information to all the slave nodes. Once all slaves have been initialized, the master begins the load generation process by starting a single slave while rest of the slaves are idling.

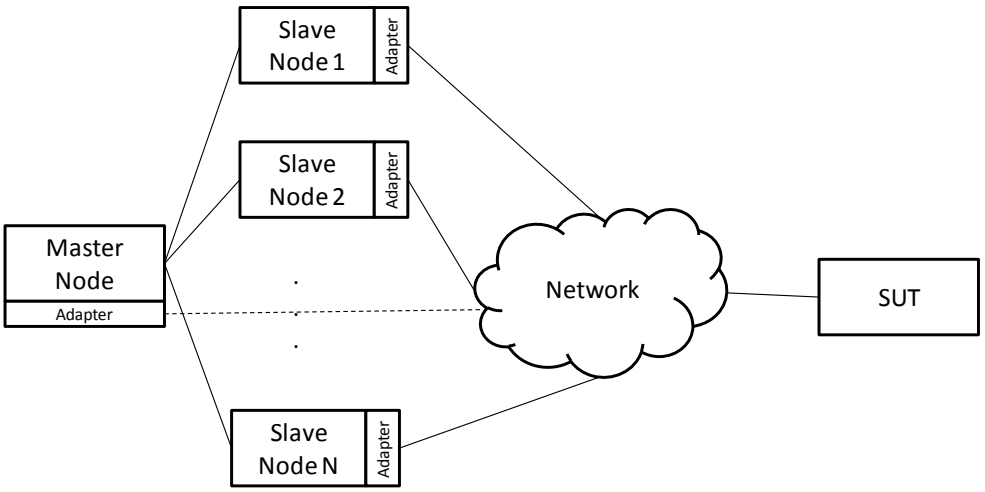


Figure 6.1: Distributed architecture of MBPeT tool

6.4.1 The Master Node

The internal architecture of the master node is shown in Figure 6.2. It contains the following components:

Core Module

The core module of the master node controls the activities of other modules as well as the flow of information among them. It initiates the different modules when their services are required. The core module takes as input the following information and distributes it among all the slave nodes:

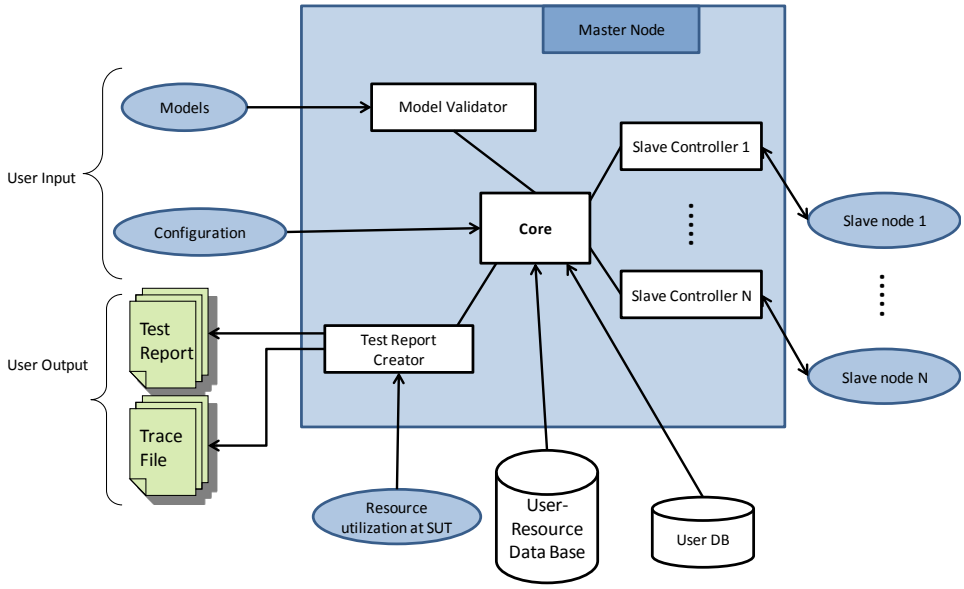


Figure 6.2: Master Node

1. *User Models:* PTA models are employed to mimic the dynamic behavior of the users. Each case-study can have multiple models to represent different types of users. User models are expressed in DOT language [5].

2. *Test Configuration:* It is a collection of different parameters, that are defined in a *Settings* file, which is a case-study specific. A *Settings* file specifies the necessary information about the case-study and this information is later used by the tool to run the experiment. There are some mandatory parameters in the *Settings* file, which have been listed below with the brief description. These parameters can also be provided as command-line arguments to the master node.

   (a) *Test duration*: It defines the duration of a test session in seconds.

   (b) *Number of users*: It specifies the maximum number of concurrent users for a test session.

   (c) *Ramp*: The ramp period is specified for all types of users. It can be defined in two ways. One way is to specify it as a percentage

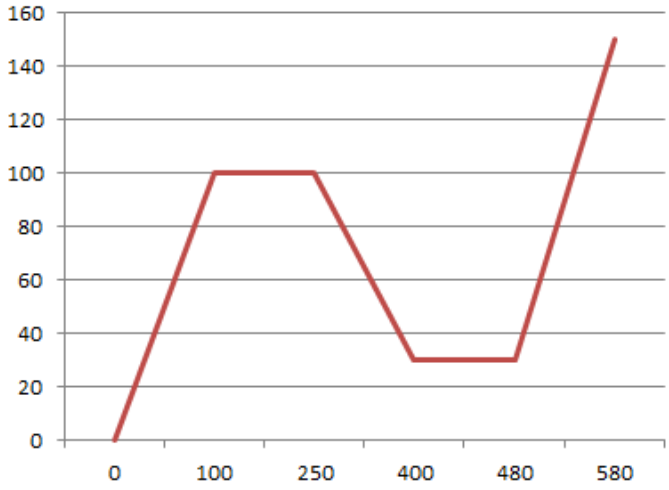


Figure 6.3: Example ramp function

of the total test duration. For example, if the objective of the experiment is to achieve the given number of concurrent users within the 80% of total test duration, then the ramp value would be equal to 0.8. Then, the tool would increase the number of users at a constant rate, in order to achieve the given number of concurrent users within the ramp period.

The ramp period can also be defined as an array of tuples. For instance the ramp function depicted in Figure 6.3, as illustrated in the Listing 6.1. A pair value is referred to as a *milestone*. The first integer in a milestone describes the time duration in seconds since the experiment started and the second integer states the target number of concurrent users at that moment. For example, the fourth milestone in the Listing 6.1, that is (400, 30), indicates that at 400 seconds the number of concurrent users should be 400, and thus starting from the previous milestone (100, 30) the number of concurrent users should drop linearly in the interval 250-400 seconds. Further, a ramp period may consist of several milestones depending upon the experiment design. The benefit of defining the ramp period in this way is that the number of concurrent users could increase and decrease during the test session.

Listing 6.1: Ramp section of Settings file

```
#================ Ramp Period ================
ramp_list = [(0, 0), (100, 100), (250, 100),
(400, 30),(480, 30), (580, 150), ... ]
```

(d) *Monitoring interval*: It specifies how often a slave node should check and report its own local resource utilization level for saturation.

(e) *Resource utilization threshold*: It is a percentage value which defines the upper limit of local resource load at the slave node. A slave node is considered to be saturated if the limit is exceeded.

(f) *Models folder*: A path to a folder which contains all the user models.

(g) *Test report folder*: The tool will save the test report at this given path.

In addition to mandatory parameters, the *Settings* file can contain other parameters, which are related to a particular case-study only. For example, if a SUT is a web server then the IP address of the web server would be an additional parameter in the *Settings* file.

3. *Adapter:* This is a case-study specific module which is used to communicate with SUT. This module translates each action interpreted from the PTA model into a form that is understandable by the SUT, for instance a HTTP request. It also parses the response from the SUT and measures the response time.

4. *Number of Slaves:* This number tells the master node how many slave nodes that are participating in the test session.

Two test databases are used by MBPeT: a user database and a user resource database. The user database contains all the information regarding users such as usernames, passwords or name spaces. In certain cases, the current state of the SUT must be captured, in order to be able to address at load generation time data dependencies between successive requests. As such, the user resource database is used to store references to the resources (e.g. files) available on the SUT for different users. The core module of the master node uses an instance of the test adapter to query the SUT and save that data in the user resource database.

Further, the core module remotely controls the Dstat[1] tool on SUT via SSH protocol. Dstat is a tool that provides detailed information about the system resource utilization in real-time. It logs the system resources utilization information after every specific time interval, one second by default. The delay between each update is specified in the command along with the names of resources to be monitored. This tool creates a log file in which it appends

[1]http://dag.wieers.com/home-made/dstat/

a row of information for each resource column after every update. The log file generated by the Dstat tool is used as basis for generating the test report, including graphs on how SUT's KPIs vary during the test session.

Model Validation Module

The *Model Validator* module validates the load models. It performs different numbers of syntactic checks on all models and generates a report.This report gives error descriptions and the location in model where the error occurred. A model with syntax anomalies could lead to inconclusive results. Therefore it is important to ensure that the all given models are well-formed and no syntax mistakes have been made in implementing the models. Examples of couple of validation rules are:

- Each model should have an initial and a final state
- All transitions have either probabilities or actions
- The sum of probabilities of transitions originating from a location is 1.
- All locations are statically reachable

Slave Controller Module

For each slave node there is an instance of *SlaveController* module in the master node. The purpose of the SlaveController module is to act as a bridge between slave nodes and the core master process and to control the slave nodes until the end of the test. The benefit of this architecture is to keep the master core process light and active, and more scalable. The SlaveController communicates with master core process only in few special cases, so that the core process could perform other tasks instead of communicating with slave nodes. Moreover, it also increases the parallelism in our architecture, all the SlaveControllers and the master's core processes could execute in parallel on different processor cores. Owning to the efficient usage of available resources, the master can perform more tasks in less period of time. A similar approach has been employed at the slave node, where each user is simulated as an independent process for the performance gain.

Test Report Creation Module

This module performs two tasks: Data Aggregation and Report Creation. In the first task, it combines the test results data from all slaves into an internal representation. Further, it retrieves the log file generated by the Dstat tool

from the SUT via Secure File Transfer Protocol (SFTP). The second task of this module is to calculate different statistical indicators and render a test report based on the aggregated data.

6.4.2 The Slave Node

Slave nodes are started with one argument, the IP-address of the master node. The *Core* module opens the socket and connects to the master node at the given IP-address with the default port number. After connecting with the master node successfully, it invokes the Load Initiator module.

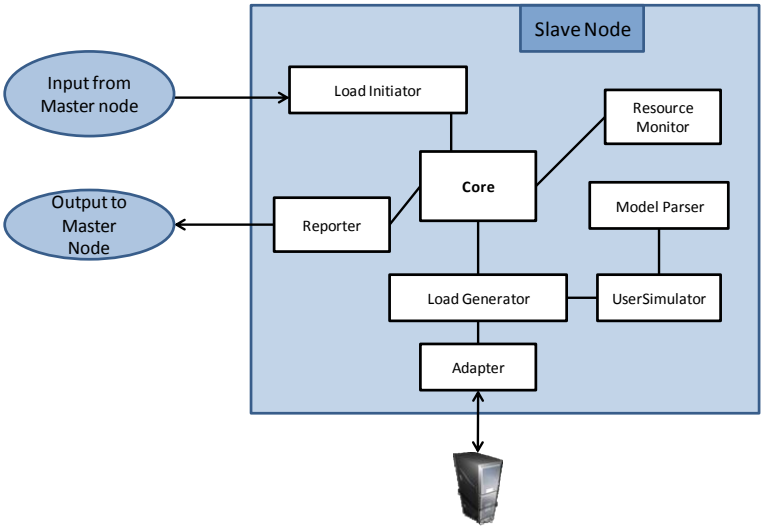


Figure 6.4: Slave Node

Load Initiation Module

The *Load Initiator* module is responsible for initializing the test setup at the slave node as well as storing the case-study and model files in a proper directory structure. It receives all the information from the master node at initialization time.

Model Parser Module

The *Model Parser* module reads the PTA model into an internal structure. It is a helper module that facilitates the UserSimulator module to perform different operations on the PTA model.

Load Generation Module

The purpose of this module is to generate load for the SUT at the desired rate, by creating and maintaining the desired number of concurrent virtual users. It uses the *UserSimulator* module to simulate virtual users where each instance of UserSimulator presents a separate user with unique user ID and session. The UserSimultor utilizes the Model Parser module to get the user's action from the user model and uses the Adapter module to perform the action. Then it waits for a specified period of time (i.e. the user think time) before performing the next action, which is chosen based on the probabilistic distribution.

Resource Monitoring Module

The *Resource Monitor* module runs as a separate thread and wakes up regularly after a specified time period. It performs two tasks every time it wakes up: 1) checks the local resource utilization level and saves the readings, 2) calculates the average of resource utilizations over a certain number of previous consecutive readings. The value obtained from the second task is compared with resource utilization threshold value, defined in the test configuration. If the calculated average is above a set threshold value of 80 percent, then it means that the slave node is about to saturate and the master will be notified. When a slave is getting saturated, its current number of generated users is kept constant, and additional slaves will be delegated to generate the more load.

Reporter Module

All the data that has been gathered during the load generation is dumped into files. The Load Generator creates a separate data file for each user; it means that the total number of simulation data files would be equal to the total number of concurrent users. In order to reduce the communication delay, all these data files are packed into a zip file, and sent to the master at the end of the test session.

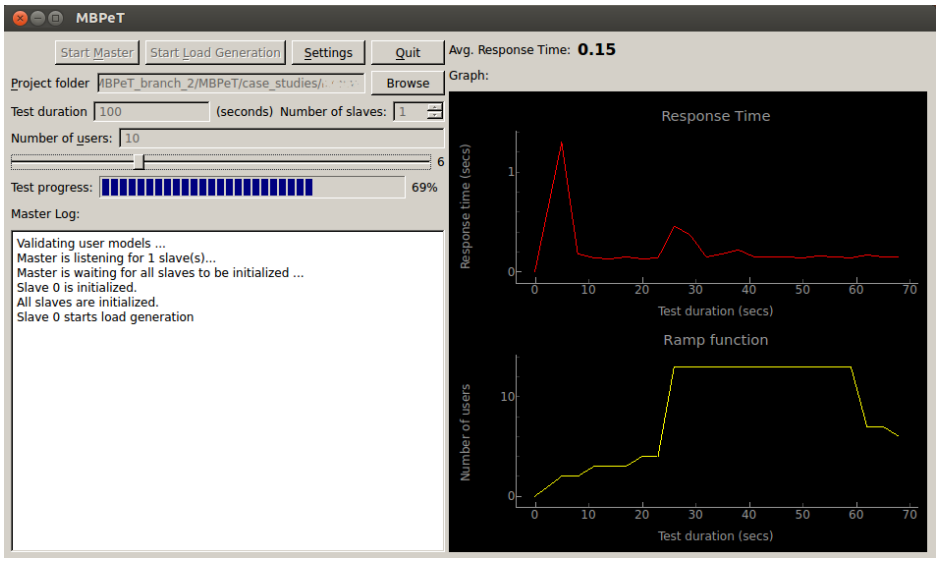


Figure 6.5: Main window of the GUI

6.4.3 Graphical User Interface

The MBPeT tool can be run both in command line and via a graphical user interface (GUI) as shown in Figure 6.5. Feature-wise the GUI is almost identical to the command-line version except for two features:

- The GUI implements the number of users as a slider function. This implies that the number of parallel user can be increased and decreased in real time using the slider, as an alternative to predefining a ramp function at beginning of the test session;
- The average response observed by all slave nodes is plotted in real-time. The response time graphs can be configured to display either one average response time plot for all actions (as currently depicted in Figure 6.5) or one average response time plot for each individual action type.

Additionally, from the GUI, one can specify basically all the test session settings previously described in Section 6.4.1

6.5 Model Creation

In this section we will introduce the load models used for generating load and describe how they are constructed. We will also in theory describe how

load is generated from these models.

6.5.1 Workload Characterization

Traditionally, performance analysis starts first with identifying key performance scenarios, based on the idea that certain scenarios are more frequent than others or certain scenarios impact more on the performance of the system than other scenarios. A performance scenario is a sequence of actions performed by an identified group of users [13]. In some cases, key performance scenarios can consist of only one action, for example "browse", in the case of a web-based system. In the case of Amazon online store, examples of key performance scenarios could be: searching for a product, then adding one or more products into the shopping cart and finally pay for them. In the first example, only one action is sent to the system, namely "browse". In the second example, several actions would have to be sent to the server, e.g. "login", "search","add-to-cart","checkout", etc.

In order to build the workload model, we start by looking and analyzing the requirements and the system specifications, respectively. During this phase we try to get an understanding of how the system is used, what are the different types of users, and what are the key performance scenarios that will impact most on the performance of the system. A user type is characterized by the distribution and the types of actions if performs.

The main sources of information for workload characterization are: Service Level Agreements (SLAs), system specifications and standards, and server execution logs [11]. By studying these sources we identify the inputs of the system with respect to types of transactions (actions), transferred files, file sizes, arrival rates, etc. following the generic guidelines discussed in [3]. In addition, we extract information regarding the KPIs, such as the number of concurrent users the system should support, expected throughput, response times, expected resource utilization demands etc. for different actions under a given load.

We use the following steps in analyzing the workload:

1. Identify the actions that can be executed against the system.
   (a) Analyze what are the required input data and output data for each action. For instance, what is the request type, its parameters, etc.
   (b) Identify dependencies between actions. For example, a user can not execute a logout action before a login action.
2. Identify what classes (types) of users execute each action

3. Identify the most relevant user types.

4. Define the distribution of actions that is performed by each user type.

5. Define an average *think time* per action for each user type.

Table 6.1 shows an example of a user type specification, its actions, action dependencies, and think time ordered in a tabular format. Based on this information we build a *workload model* described as a *probabilistic timed automata* or *PTA*.

Action	Dependency	User Type 1		User Type 2	
		Think time	Frequency	Think Time	Frequency
a_1		t_1	f_1	t_2	f_2
a_2	a_1	t_3	f_3		
a_3	a_1			t_4	f_4
a_4	a_2	t_5	f_5		
a_5	a_4	t_6	f_6	t_7	f_7
a_6	a_3			t_8	f_8

Table 6.1: Example of user types and their actions

6.5.2 Workload Modeling Using PTA

The results of the workload characterization are aggregated in a workload model similar to the one in Figure 6.6, which mimics the real workload under study. One such workload model is created for each identified user type. Basically, the model will depict the sequence of actions a user type can perform and their arrival rate, as a combination of the probability that an action is executed and the think time of the user for that action. In addition, we also identify the user types and their probabilistic distribution. A concrete example will be given in Section 6.7.

All the information that is extracted from the previous phase is aggregated in a workload model which is describes as a probabilistic timed automaton (PTA). A PTA is similar to a state machine in the sense that a PTA consists of a set of locations connected with each other via a set of transitions. However, a PTA also include the notion of time and probabilities. Time is modeled as an invariant clock constraint on transitions and increase at the same rate as real time.

A *probabilistic timed automaton* (PTA) is defined [9] as T = (L,C,*inv*, *Act*, E, δ) where:

- *a set of locations L;*

- *a finite set of clocks C;*
- *an* invariant condition *inv :* L $\rightarrow$ Z;
- *a finite set of* actions *Act;*
- *an* action enabledness function $E : L \times Act \rightarrow Z$;
- *a* transition probability function $\delta : (L \times Act) \rightarrow D(2^C \times L)$.

In the above definitions, Z is a set of clock zones. A clock zone is a set of clock values, which is a union of a set of clock regions. Δ is a probabilistic transition function. Informally, the behavior of a probabilistic timed automaton is as follows: In a certain location l, an action a can be chosen when a clock variable reaches its value with a certain probability if the action is enabled in that location l. If the action a is chosen, then the probability of moving to a new location l' is given by δ[l,a](C',l'), where C' is a particular set of clocks to be reset upon firing of the transitions. Figure 6.6 gives an example of a probabilistic timed automata.

The syntax of the automata is as follows: Every transition has an initial location and an end location. Each location is transitively connected from the initial location. The transitions can be labeled with three different values: a probability value, an action, and a clock. The *probability* indicates the chance of that transition being taken. The *action* describes what action to take when the transition is used, and the *clock* indicates how long to wait before firing the transition. Every automaton has an *end location*, depicted with a double circle, that will eventually be reached. It is possible to specify loops in the automaton. It is important to notice that the sum of the probabilities on all outgoing transitions from a given location must be equal to 1. For example, consider location *2* in Figure 6.6: for the PTA to be complete the following must apply: p1 + p2 + p3 = p4 + p5 = 1.

6.6 Performance Testing Process

In this section we describe the performance testing process. Figure 6.7 shows the three steps involved in the process. In the following, we will discuss the three steps in more detail.

6.6.1 Test Setup

Every test run starts with a test setup. In each test setup, there is one master node that carries out the entire test session and generates a report. The

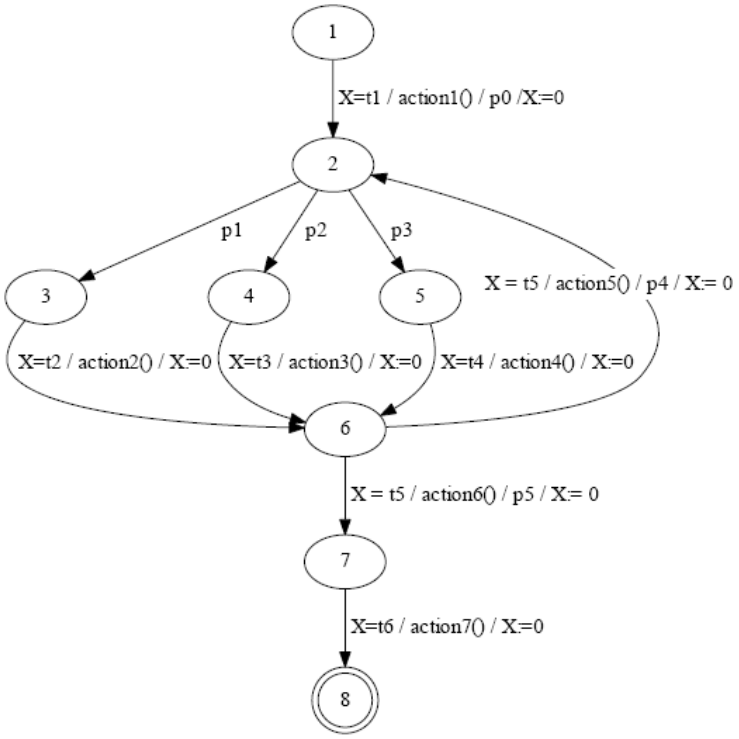


Figure 6.6: Example of a probabilistic timed automaton.

user only interacts with the master node by initializing it with the required parameters (mentioned in the Section 6.4.1) and getting the test report at the end of the test run. The parameter given to the master is the project folder. This folder contains all the files needed for load generation, such as the adapter code, the settings file (if command line mode is used) and other user specific files.

The adapter file and the settings file are the most important. The adapter files explains how the abstract actions found in the load models are translated to concrete actions. The settings file contain information about the test session, such as the location of the load models, IP-address to the SUT, the ramp function, test duration, etc. The same information can also be set from the GUI via the *Settings* button, see Figure 6.8. In here, the user is required to enter the same information as given in the settings file. Additionally, the path to the adapter file and the load models have to be given.

As one may notice in Figure 6.8, the user has the option of defining an average *think time* for the models and its *standard deviation*. If these options are used, the individual think time specified in the models for each action

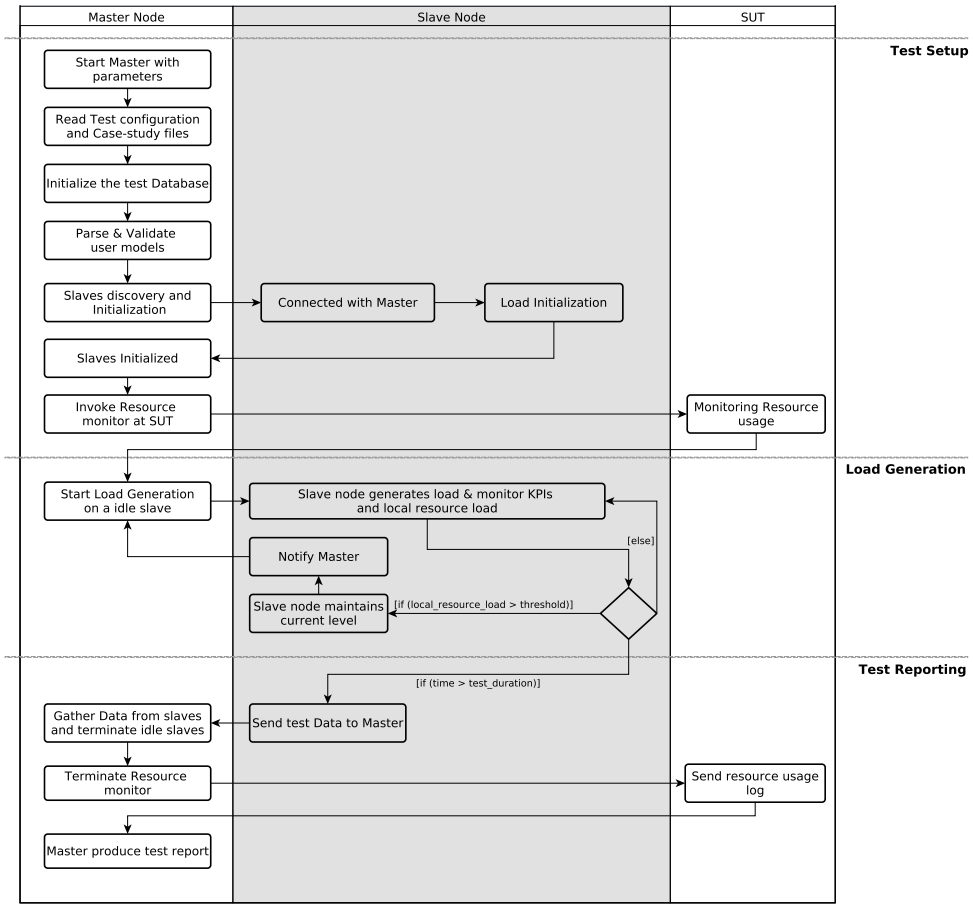


Figure 6.7: MBPeT tool activity diagram

will be ignored and the one specified in the GUI will be used.

Once the required information has been given, the master node sets up the test environment. After that, it invokes the *Model Validator*. This module validates the syntax of user models. If the validation fails, it gives the user a choice whether the user wants to continue or not to load generation. If the user decides to continue or the validation was successful, then the master enters into the next phase.

6.6.2 Load Generation

Load is generated for the models based on the same principles as described in section 6.5.2. The load generation is based on a deterministic choice with a probabilistic policy. This introduces certain randomness into the test process

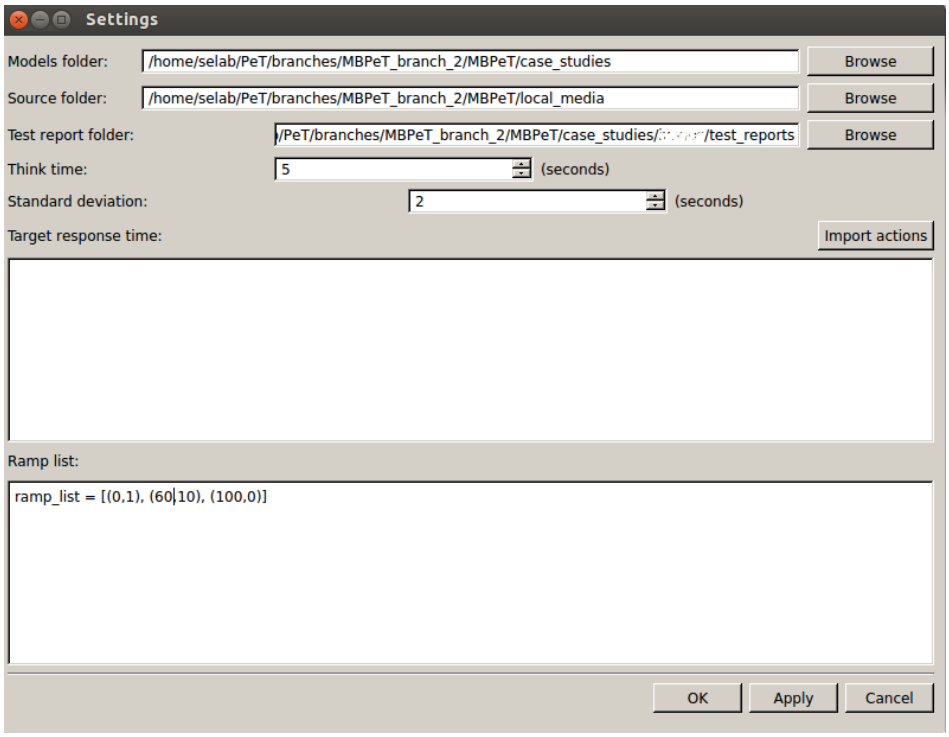


Figure 6.8: Settings window of the GUI

and that can be useful for uncovering certain sequences of actions which may have a negative impact of the performance. Such sequences would be difficult or maybe impossible to discover if static test scripts are used, where a fixed order of the actions is specified, and repeated over and over again. Every PTA has an *exit* location which will eventually be reached. By modifying the probability for the *exit* action, it is possible to adjust the length of the test.

The attributes of PTA models make them a good candidate for modeling the behavior of VUs, which imitate the dynamic behavior of real users. Actions in the PTA model corresponds to an action which a user can send to the SUT and the clocks present the user *think time*. In our case, the PTA formalism is implemented using the DOT notation.

Load is generated from these models by executing an instance of the model for every simulated VU. Whenever a transition with an action is fired, that action is translated by the MBPeT tool and sent to the SUT. This process is repeated and run in parallel for every simulated user throughout the whole test session. During load generation, the MBPeT tool monitors the SUT the whole time.

6.6.3 Test Reporting

After each test run the MBPeT tool generates a test report based on the monitored data. It is the slave nodes that are responsible for the monitoring and they report the values back to the master node which later creates the report.

Every slave node will monitor the communication with the SUT and collecting the data needed for test report. The slave node will start a timer every time and action is sent to the system. When a response is received, the timer is stopped and the response code together with the action name and response time is stored. This data is later sent to the master node which will aggregate the data and produce a report.

The slave node will also monitor its own resources so it does not get saturated and becomes the bottleneck during load generation. The slave node monitors is own CPU, memory, and disk utilization and sends the information to the master node. The master node the data is plotted in graphs and included in the test report.

It is the test report creation module of the master node that is responsible for creating test report. This module performs two tasks: aggregating data received from the slave nodes and creating a test report. Data aggregation consists of combining data received from the slave nodes together into and internal representation. Based on the received data, different kinds of statistical values are computer, e.g. mean and max response times, throughput, etc. Values such as response time and throughput plotted as graphs so the tester can see how the different values varies over time. Figures of the test report will later be shown throughout Section 6.7.

The final task of the test report creation module is to render all the values and graphs into a report. The final report is rendered as a HTML document.

6.7 Experiments

In this section we will describe a set of experiments carried out with the MBPeT tool on a case study. The system tested in the case study is an HTTP based auction web service.

6.7.1 YAAS

YAAS is a web application and a web service for creating and participating in auctions. An auction site is a good example of a service offered as a web application. It facilitates a community of users interested in buying or selling diverse items, where any user including guest user can view all the auctions

and all authenticated users, except seller of an item, can bid on the auction against other users.

The web application is implemented in Python language using the Django[2] web-framework. In addition to HTML pages, YAAS also has a RESTful [10] web service interface. The web service interface has various APIs to support different operations, including:

Browse API It returns the list of all active auctions.

Search API It allows to search auctions by title.

Get Auction This API returns an auction against the given Auction-ID.

Bids It is used to the get the list of all the bids have been made to a particular auction.

Make Bid Allows and authenticated user to place a bid on a particular auction.

6.7.2 Test Architecture

A setup of the test architecture can be seen in Figure 6.9. The server runs an instance of the YAAS application on top of an Apache web server. All nodes (master, slaves, and the server) feature an 8-core CPU, 16GB of memory, 1Gb Ethernet, 7200 rpm hard drive, and Fedora 16 operating system. The nodes were connected via a 1Gb ethernet over which the data were sent.

A populator script is used to generate input data (i.e., populate the test databases) on both the client and server side, before each test session. This ensures that the test data on either sides is consistent and easy to rebuild after each test session.

6.7.3 Load Models

The test database of the application is configured with a script to have 1000 users. Each user has exactly one auction and each auction has one starting bid.

In order to identify the different type of users for the YAAS application, we have used the AWStats[3] tool. This tool analyzes the Apache server access logs to generate a report on the YAAS application usage. Based on that report, we discovered three types of users; aggressive, passive and non-bidder.

[2]https://www.djangoproject.com/

[3]http://awstats.sourceforge.net

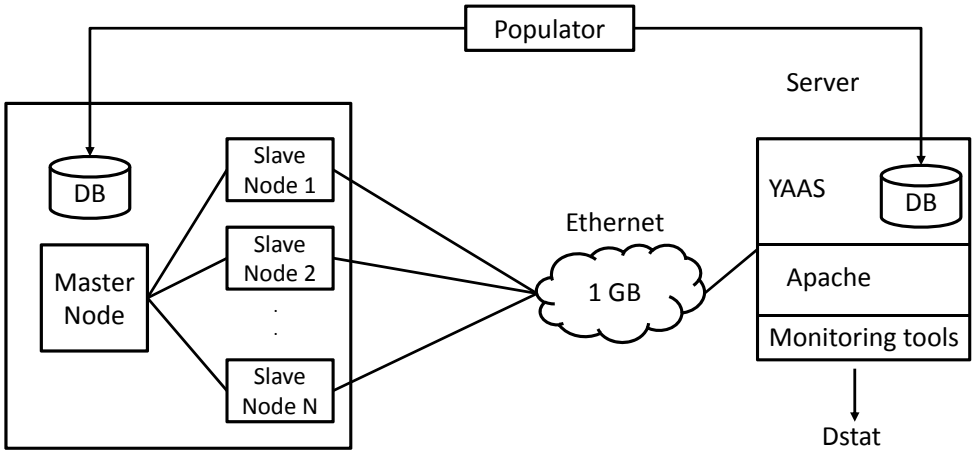


Figure 6.9: A caption of the test architecture

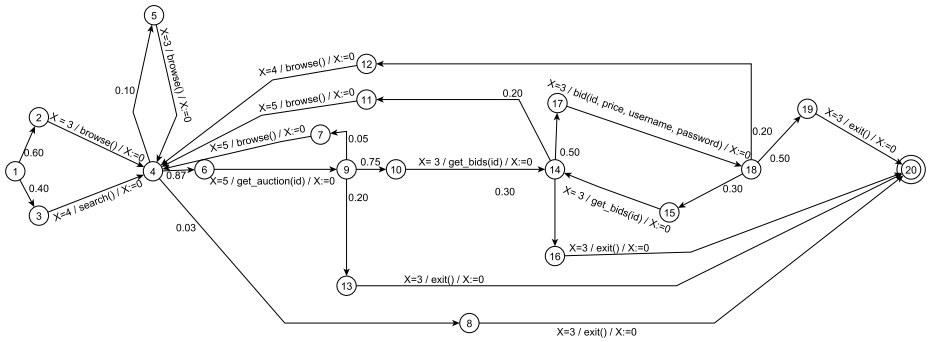


Figure 6.10: Aggressive User type model

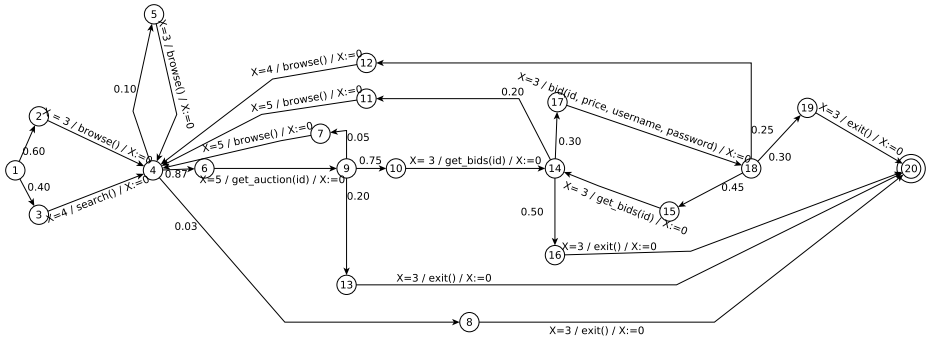


Figure 6.11: Passive User type model

Action	Dependency	Aggressive User		Passive User		Non-Bidder User	
		Think time	Frequency	Think Time	Frequency	Think Time	Frequency
search()		4	0,40	4	0,40	4	0,40
browse()		3	0,60	3	0,60	3	0,60
browse()	browse(),search()	5	0,10	3	0,10	3	0,10
get_auction()	browse(),search()	5	0,87	5	0,87	5	0,87
exit()	browse(),search()	3	0,03	3	0,03	3	0,03
browse()	get_auction()	5	0,05	5	0,05	5	0,05
get_bids()	get_auction()	3	0,75	3	0,75	3	0,75
exit()	get_auction()	3	0,20	3	0,20	3	0,20
browse()	get_bids()	5	0,20	5	0,20	5	0,60
bid()	get_bids()	3	0,50	3	0,30		
exit()	get_bids()	3	0,30	3	0,50	3	0,40
get_bids()	bid()	3	0,30	3	0,45		
browse()	bid()	4	0,20	4	0,25		
exit()	bid()	3	0,50	3	0,30		

Table 6.2: Think time and distribution values extracted from the AWStats report

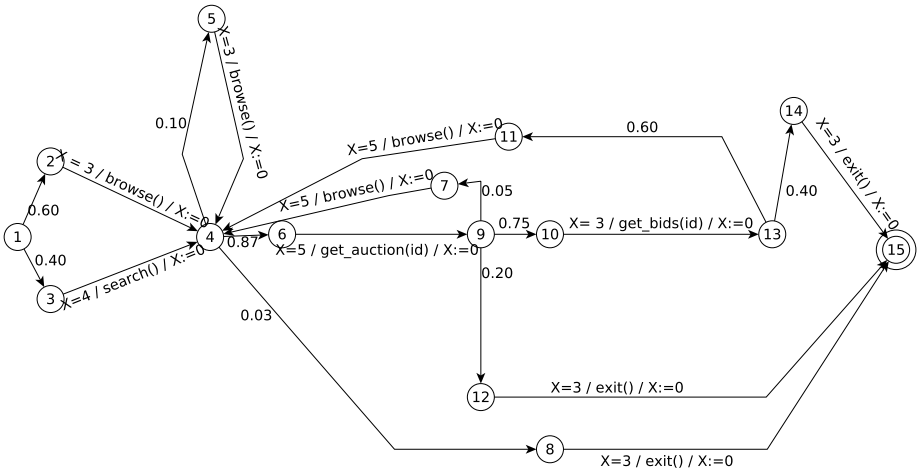


Figure 6.12: Non-bidder User type model

Table 6.2 shows the think time and distribution of actions for the three different types of users.

For each user type, a load model was created as describe in section 6.5. The aggressive type (Figure 6.10) of users describes those users, who make bids more frequently as compared to other types of users. The passive users (Figure 6.11) are less frequent in making bids, see for instance the locations 14 or 18 in the referred figures. The third type of users are only interested in browsing and searching for auctions instead of making any bids and are known as non-bidders (Figure 6.12). The root model of the YAAS application, shown in Figure 6.13, describes the distribution of different user types.

Based on the AWStats analysis, we determined that the almost 30% of total users who visited the YAAS, were very frequently in making bids, whereas rest of 50% users made bids occasionally. The rest of the users were not interested in making bids at all. This distribution is depicted by the model in Figure 6.13.

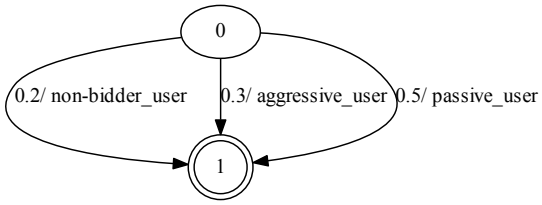


Figure 6.13: YAAS Root model

The models of all these user types were provided to the MBPeT tool to simulate them as virtual users. For example, the model of an *aggressive user type*, shown in Figure 6.10, shows that the user will start from the location *1*, and from this location the user will select either *browse* or *search* action based on a probabilistic choice. Before performing the action, the slave will wait for the think time corresponding to the selected action. Eventually, the user will reach the final location (i.e. location *20*) by performing the *exit* action and terminate the current user session. Similarly, the other models of *passive* and *non-bidder* user type have the same structure but with different probabilities and distribution of actions.

6.7.4 Experiment 1

The goal of this experiment was to set the target response time for each action and observe at what point the average response time of the action exceed the target value. The experiment ran for 20 minutes. The maximum number of concurrent users was set to 300 and the ramp up value was 0.9 that the tool would increase the number of concurrent users with the passage of time to achieve the value of 300 concurrent users when the 90% of test duration time has been passed.

The resulting test report has various sections, where each section presents the different perspective of the results. The first section, shown in Figure 6.14, contains the information about the test session including, test started time, test duration, target number of concurrent of users, etc. The *Total*

###################### Master Stats ######################

This test was executed at: 2013-07-01 16:54:47
Duration of the test: 20 min
Target number of concurrent users: 300
Total number of generated users: 27536
Measured Request rate (MRR): 27.68 req/s
Number of NON-BIDDER_USER: 6296 (23.0)%
Number of AGGRESSIVE_USER: 9087 (33.0)%
Number of PASSIVE_USER: 12153 (44.0)%
Average number of action per user: 91 actions

Figure 6.14: Test Report 1 - Section 1: General information

######## AVERAGE/MAX RESPONSE TIME per METHOD CALL #########

	NON-BIDDER_USER (23.0 %)		PASSIVE_USER (44.0 %)		AGGRESSIVE_USER (33.0 %)	
Method Call	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
GET_AUCTION(ID)	3.04	23.95	2.85	23.67	2.93	24.71
BROWSE()	5.44	21.25	5.66	21.7	5.68	21.29
GET_BIDS(ID)	3.59	27.37	3.63	25.8	3.65	24.87
BID(ID,PRICE,USERNAME,PASSWORD)	0.0	0.0	8.26	33.44	8.11	36.84
SEARCH(STRING)	3.36	12.86	3.26	15.84	3.47	15.79

Figure 6.15: Test Report 1 - Section 2: Average and Maximum response time of SUT per action or method call

number of generated users in the report describes that the tool had simulated 27536 numbers of virtual users. The *Measured Request Rate (MRR)* depicts the average number of requests per second which were made to the SUT during the load generation process. Moreover, it also shows the distribution of total number of user generated which is very close to what we have defined in the root model (Figure 6.13). This section is useful to see the summarized view of the entire test session.

In the second section of the test report, we could observe the SUT performance for each action separately, and identify which actions have responded with more delay than the others, and which actions should be optimized to increase the performance of the SUT. As from the table in Figure 6.15, it appears that the action *BID(ID, PRICE, USERNAME, PASSWORD)* has larger average and maximum response time than the other actions. The *non-bidder* users do not perform the *BID* action that is why we have zero response time in the column of *NON-BIDDER_USER* against the *BID* action.

Section three (shown in Figure 6.16) of the test report presents a comparison of the SUTs desired performance against the measured performance. As we had defined the target response time for each action in the test config-

AVERAGE/MAX RESPONSE TIME THRESHOLD BREACH per METHOD CALL

Action	Target Response Time		NON-BIDDER_USER		PASSIVE_USER		AGGRESSIVE_USER		Verdict
	Average (secs)	Max (secs)	Average users (secs)	Max users (secs)	Average users (secs)	Max users (secs)	Average users (secs)	Max users (secs)	Pass/Fail
GET_AUCTION(ID)	2.0	4.0	70 (251)	84 (299.0)	70 (251)	95 (341.0)	70 (250)	95 (341.0)	Failed
BROWSE()	4.0	8.0	84 (299)	97 (345.0)	84 (299)	113 (403.0)	84 (299)	113 (403.0)	Failed
GET_BIDS(ID)	3.0	6.0	84 (298)	112 (402.0)	83 (296)	112 (402.0)	96 (344)	112 (401.0)	Failed
BID(ID,PRICE,USERNAME,PASSWORD)	5.0	10	Passed	Passed	97 (346)	113 (405.0)	112 (402)	135 (483.0)	Failed
SEARCH(STRING)	3.0	6	95 (341)	134 (479.0)	96 (342)	112 (402.0)	83 (296)	133 (476.0)	Failed

Figure 6.16: Test Report 1 - Section 3: Average and Maximum response time of SUT per action or method call

uration, in this section we could actually observe how many concurrent users were active when the target response time was breached. The table in this section allows us to estimate the performance of current system's implementation. For instance, the target average response time for the *GET_AUCTION* action was breached at 250 seconds for the *aggressive* type of users, when the number of concurrent users was 70. Further, this section demonstrates that the SUT can only support up to 84 concurrent users before it breaches the threshold value of 3 seconds for *GET_BIDS* action for the *passive* type of users. In summary, all the actions in Figure 6.16 have breached the target response time except the *BID* action in *NON-BIDDER_USER* column because *non-bidder* users do not bid.

Figures 6.17 and 6.18 display the resource load at the SUT during load generation. These graphs are very useful to identify which resources are being utilized more than the others and limiting the performance of SUT. For instance, it can be seen from Figure 6.17 that after 400 seconds the CPU utilization was almost equal to 100% for the rest of the test session, it means that the target web application is CPU-intensive, and it might be the reason of large response time.

Figure 6.19 illustrate that the response time of each action for the aggressive user type increases proportionally to the number of concurrent users. The figure also points out which actions response time is increasing much faster than the other actions and require optimization. Similar patterns was observed for the two other user types: passive users and non-bidder, respectively.

For example the response time of action *BID(ID, PRICE, USERNAME, PASSWORD)* for *aggressive* and *passive* user types increases more rapidly than the other actions. It might be because the *BID* action involves a write operation and in order to perform a write operation on the database file, the

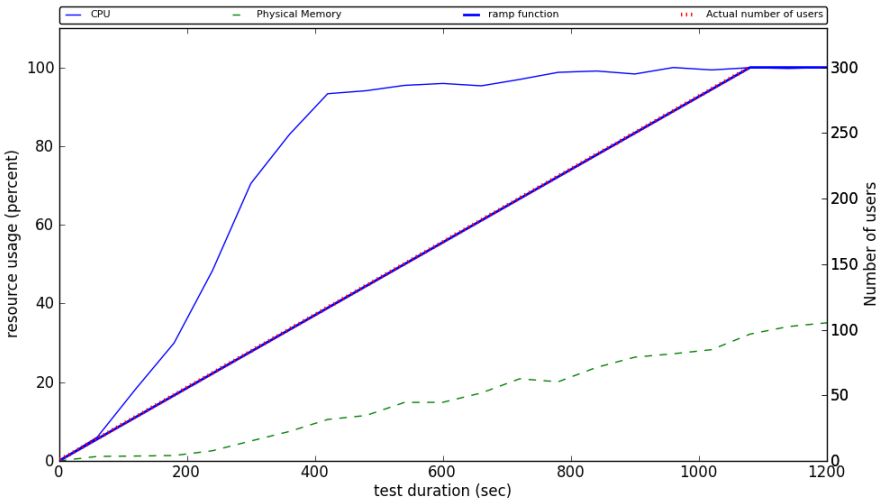


Figure 6.17: Test Report 1 - SUT CPU and memory utilization

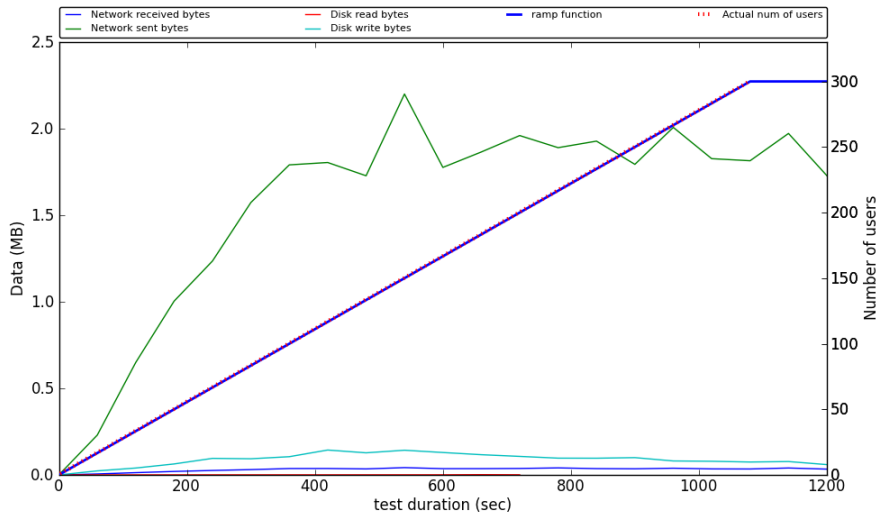


Figure 6.18: Test Report 1 - SUT network and disk utilization

SQLite[4] database has to deny the all new access requests to the database and wait until all previous operations (including read and write operations) have been completed.

Section four of the test report provides miscellaneous information about

[4]http://www.sqlite.org/

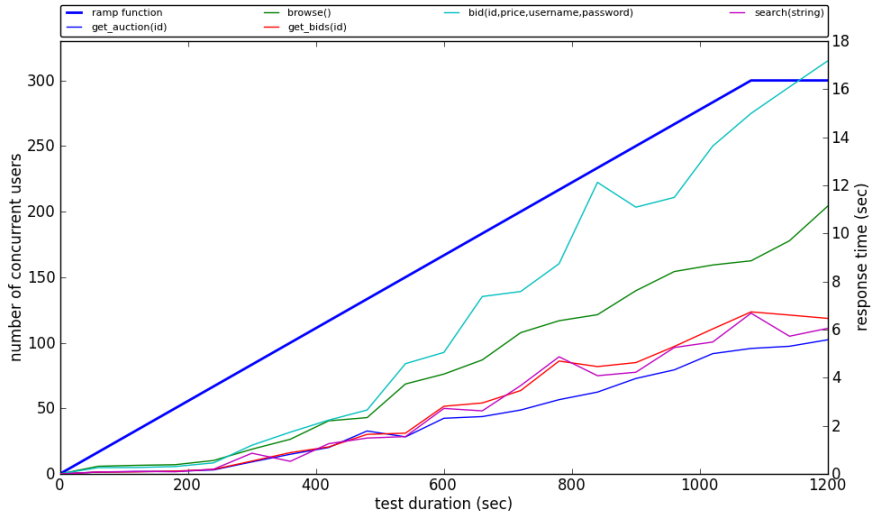


Figure 6.19: Test Report 1 - Response time of aggressive user type per action

the test session. For example, the first erroneous response was recorded at 520 seconds (according to Figure 6.20) and at that time the tool was generating load at the maximum rate, that is 1600 actions/seconds, shown in Figure 6.21. Similarly, Figure 6.20 displays that there was no error until the number of consecutive users exceeded 150, after this point errors began to appear and increased steeply proportional to the number of consecutive users.

A further deep analysis of the test report showed that the database could be the bottleneck. Owning to the fact a *sqlite* database has been used for this experiment, the application has to block the entire database before something can be written to it. It could explain the larger response time of *BID* actions compared to other actions. This is because the web application had to perform a write operation to the database in order to execute the *BID* action. Further, before each write operation, *sqlite* creates a rollback journal file, an exact copy of original database file, to preserve the integrity of database [17]. This could also delay the processing of a write operation and thus cause a larger response time.

6.7.5 Experiment 2

In the second experiment, we wanted to verify the hypothesis, which we proposed in the previous experiment: *database could be the performance bottleneck*. We ran the second experiment for 20 minutes with the same test

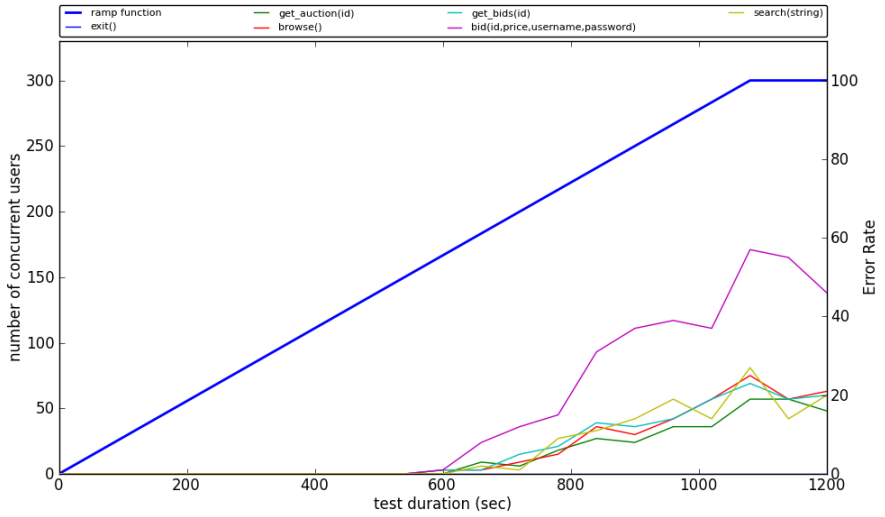


Figure 6.20: Test Report 1 - Error rate

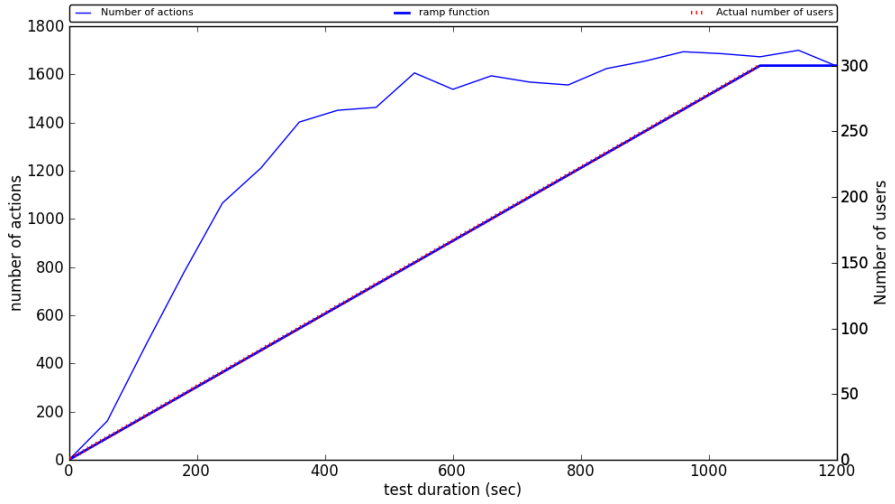


Figure 6.21: Test Report 1 - Average number of actions

configuration of the previous experiment. However, we did make one modification in the architecture. In the previous experiment, the *SQLite 3.7* was used as database server, but in this experiment, it was replaced by the *PostgreSQL 9.1* [5]. The main motivating factor of using the PostgreSQL

[5]http://www.postgresql.org

#################### Master Stats ########################

This test was executed at: 2013-07-01 17:37:38
Duration of the test: 20 min
Target number of concurrent users: 300
Total number of generated users: 35851
Measured Request rate (MRR): 39.21 req/s
Number of AGGRESSIVE_USER: 11950 (33.0)%
Number of NON-BIDDER_USER: 7697 (21.0)%
Number of PASSIVE_USER: 16204 (45.0)%
Average number of action per user: 119 actions

Figure 6.22: Test Report 2 - Section 1: global information

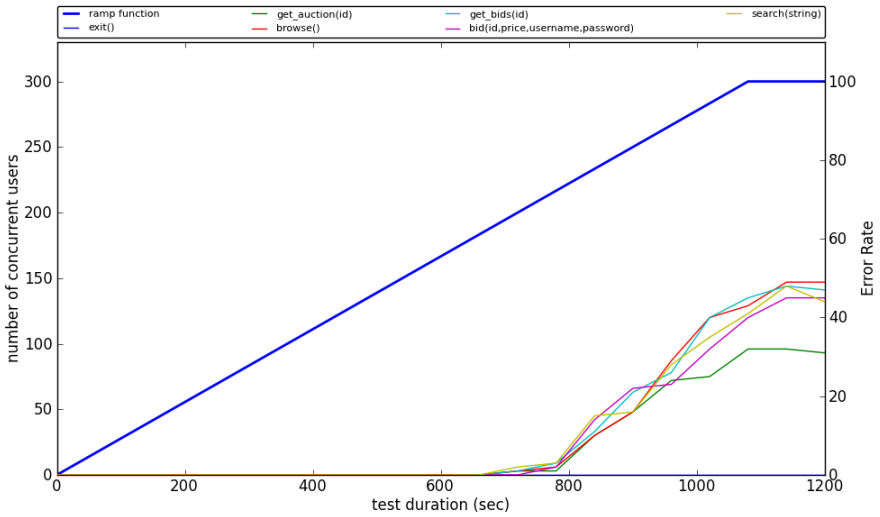


Figure 6.23: Test Report 2 - Error rate

database is that it supports the better concurrent access to the data than the SQLite. The PostgreSQL database uses the Multiversion Concurrency Control (MVCC) model instead of simple locking. In MVCC, different locks are acquired for the read and write operations, it means that the both operations can be performed simultaneously without blocking each other [14].

In the section 1 of Test report 2 (Figure 6.22) shows that the *Measured Request Rate (MRR)* increased by 42%. Additionally, each user performed averagely 30% more actions in this experiment.

Similarly in the second section (Figure 6.24), the average and maximum response time of all action decreased by almost 47%. Moreover, the error rate section (Figure 6.23) depicts that there was no error until the number

######## AVERAGE/MAX RESPONSE TIME per METHOD CALL ##########

	AGGRESSIVE_USER (33.0 %)		PASSIVE_USER (45.0 %)		NON-BIDDER_USER (21.0 %)	
Method Call	Average (sec)	Max (sec)	Average (sec)	Max (sec)	Average (sec)	Max (sec)
GET_AUCTION(ID)	1.18	15.58	1.1	15.95	1.25	15.8
BROWSE()	4.99	23.61	5.13	23.47	5.23	23.6
GET_BIDS(ID)	1.51	15.25	1.54	15.56	1.63	15.02
BID(ID,PRICE,USERNAME,PASSWORD)	3.25	18.65	3.25	18.37	0.0	0.0
SEARCH(STRING)	1.48	14.66	1.54	14.83	1.43	15.43

Figure 6.24: Test Report 2 - Section 2: Average and Maximum response time of SUT per action or method call

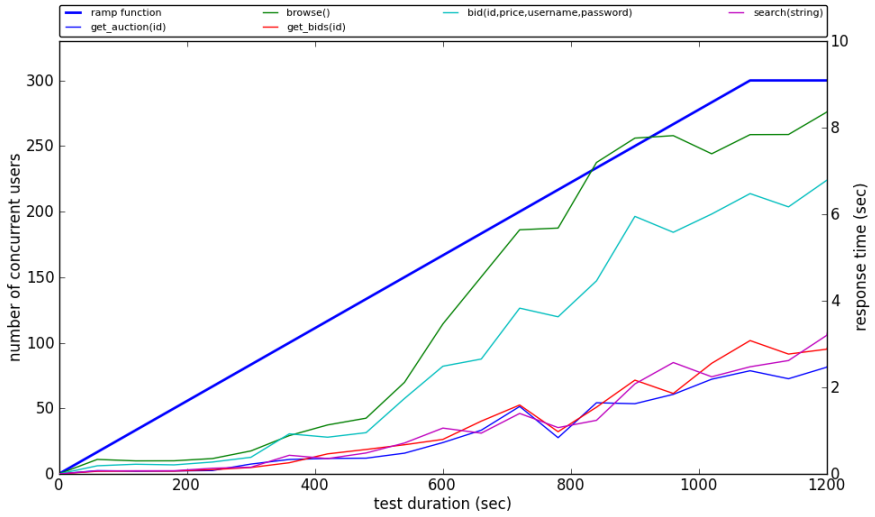


Figure 6.25: Test Report 2 - Response time of aggressive user type per action

of concurrent users was below 182, that is 21% more users than the last experiment.

Figure 6.25 shows that the response time of aggressive type of users is decreased by 50% approximately in comparison with the previous experiment in Figure 6.19. In summary, all of these indicators suggest significant improvement in the performance of SUT.

6.8 Conclusions

In this chapter, we have presented a tool-supported approach for model-based performance testing. Our approach uses PTA models to specify the probabilistic distribution of user types and of actions that are executed against

the system.

The approach is supported by the MBPeT tool, which has a distributed scalable architecture, targeted to cloud-based environments allowing it to generate load at high rates. The tool generates load in online mode and monitors different KPIs including the resource utilization of the SUT. It can be run both in command line and in GUI mode, respectively. The former facilitates the integration of the tool in automated test frameworks, whereas the latter allows the user to interact with the SUT and visualize in real-time its performance depending on the number of concurrent users.

Using our modeling approach, the effort necessary to create and update the user profiles is reduced. The adapter required to interface with the SUT has to be implemented only once and then it can be reused. As shown in the experiments, the tool allows quick exploration of the performance space by trying out different load mixes. In addition, preliminary experiments have shown that the synthetic load generated from probabilistic models has in general a stronger impact on the SUT compared to static scripts.

We have also showed that the tool us sufficient enough in finding performance bottlenecks and that the tool can handle large amounts of parallel virtual users. The tool benefits from its distributed architecture in the sense that it can easily be integrated in a cloud environment where thousands of concurrent virtual users need to be simulated.

Future work will be targeted towards improving the methods for creating the user profiles from historic data and providing more detailed analysis of the test results. So far, the MBPeT tool has been used for testing web services however, we plan also to address also web applications, as well as other types of communicating systems.

References

[1] Ashlish Jolly. *Historical Perspective in Optimising Software Testing Efforts*. 2013. URL: `http://www.indianmba.com/Faculty\_Column/FC139/fc139.html`.

[2] C. Barna, M. Litoiu, and H. Ghanbari. "Model-based performance testing (NIER track)". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 872–875. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985930`.

[3] M. Calzarossa, L. Massari, and D. Tessera. "Workload Characterization Issues and Methodologies". In: *Performance Evaluation: Origins and Directions*. London, UK, UK: Springer-Verlag, 2000, pp. 459–481. ISBN: 3-540-67193-5.

[4] G. Denaro, A. Polini, and W. Emmerich. "Early performance testing of distributed software applications". In: *Proceedings of the 4th international workshop on Software and performance*. WOSP '04. Redwood Shores, California: ACM, 2004, pp. 94–103. ISBN: 1-58113-673-0. DOI: 10.1145/974044.974059.

[5] E. Gansner, E. Koutsofios, and S North. *Drawing draphs with dot*. Online at http://www.graphviz.org/Documentation/dotguide.pdf. 2006. URL: http://www.graphviz.org/Documentation/dotguide.pdf.

[6] Hewlett-Packard. *httperf*. retrieved: October, 2012. URL: http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt.

[7] HP. *HP LoadRunner*. 2013. URL: http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451\#.URz7wqWou8E.

[8] ITEA 2. *ITEA 2 D-MINT project result leaflet: Model-based testing cuts development costs*. 2013. URL: http://www.itea2.org/project/result/download/result/5519?file=06014_D_MINT_Project_Leaflet_results_oct_10.pdf.

[9] M. Jurdziński et al. "Concavely-Priced Probabilistic Timed Automata". In: *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*. Ed. by M. Bravetti and G. Zavattaro. Vol. 5710. LNCS. Springer, 2009, pp. 415–430.

[10] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media. 2007.

[11] D. A. Menasce and V. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130659037.

[12] D. Mosberger and T. Jin. "httperfa tool for measuring web server performance". In: *SIGMETRICS Perform. Eval. Rev.* 26.3 (Dec. 1998), pp. 31–37. ISSN: 0163-5999. DOI: 10.1145/306225.306235. URL: http://doi.acm.org/10.1145/306225.306235.

[13] D. C. Petriu and H. Shen. "Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications". In: Springer-Verlag, 2002, pp. 159–177.

[14] PostgreSQL. *Concurrency Control*. retrieved: March, 2013. URL: http://www.postgresql.org/docs/9.1/static/mvcc-intro.html.

[15] G. Ruffo et al. "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance". In: *Network Computing and Applications, IEEE International Symposium on* 0 (2004), pp. 77–86. DOI: http://doi.ieeecomputersociety.org/10.1109/NCA.2004.1347765.

[16] M. Shams, D. Krishnamurthy, and B. Far. "A model-based approach for testing the performance of web applications". In: *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*. Portland, Oregon: ACM, 2006, pp. 54–61. ISBN: 1-59593-584-3. DOI: http://doi.acm.org/10.1145/1188895.1188909.

[17] SQLite. *File Locking And Concurrency In SQLite Version 3*. retrieved: March, 2013. URL: http://www.sqlite.org/lockingv3.html.

[18] Sun. *Faban Harness and Benchmark Framework*. 2013. URL: http://java.net/projects/faban/.

[19] The Apache Software Foundation. *Apache JMeter*. Retrieved: October, 2012. URL: http://jmeter.apache.org/.

Paper VII

An Automated Approach for Creating Workload Models From Server Log Data

Fredrik Abbors, Tanwir Ahmad, and, Dragos Truscan

Originally published *2014 Proceeding of 9th International Conference on Software Engineering and Applications.* SCITEPRESS. August 2014, Vienna, Austria.

An Automated Approach for Creating Workload Models From Server Log Data

Fredrik Abbors, Dragos Truscan, Tanwir Ahmad
Department of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5 A, Turku, Finland
{*fredrik.abbors, dragos.truscan, tanwir.ahmad*}*@abo.fi*

Keywords: Workload model generation, Log file analysis, Performance testing, Probabilistic Timed Automata

Abstract: We present a tool-supported approach for creating workload models from historical web access log data. The resulting workload models are stochastic, represented as Probabilistic Timed Automata (PTA), and describe how users interact with the system. Such models allow one to analyze different user profiles and to mimic real user behavior as closely as possible when generating workload. We provide an experiment to validate the approach.

1 INTRODUCTION

The primary idea in performance testing is to establish how well a system performs in terms of responsiveness, stability, resource utilization, etc, under a given synthetic workload. The synthetic workload is usually generated from some kind of workload profile either on-the-fly or pre-generated. However, Ferrari states that synthetic workload should mimic the expected workload as closely as possible (Ferrari, 1984), otherwise it is not possible to draw any reliable conclusions from the results. This means that if load is generated from a workload model, then the model must represent the real-world user behavior as closely as possible. In addition, Jain points out that one of the most common mistakes in load testing is the use of unrepresentative load (Al-Jaar, 1991).

There already exists a broad range of well established web analytics software both as open source (Analog, AWStats, Webalyzer), proprietary (Sawmill, NetInsight, Urchin), as well as web hosted ones (Google Analytics, Analyzer, Insight). All these tools have different pricing models and range from free up to several hundred euros per month. These tools provide all kinds of information regarding the user clients, different statistics, daily number of visitors, average site hits, etc. Some tools can even visualize paths that visitors take on the site. However, this usually requires a high-priced premium subscription. What the above tools do not provide is a deeper classification of the users or even a artefact that can directly be used for load testing. Such an artefact, based on real user data, would be the ideal source for generating synthetic load in a performance testing environment. Instead, the performance tester have to interpret the provided information and build his own artefact, from where load is generated. Automatically creating this artefact would also significantly speed up the performance testing process by removing the need of manual labour, and thus saving time and money.

This paper investigates an approach for automatically creating a workload model from web server log data. More specifically, we are targeting HTTP-based systems with *RESTful* (Richardson and Ruby, 2007) interfaces. The proposed approach uses the K-means algorithm to classify users into groups based on the requested resources and a probabilistic workload model is automatically built for each group.

The presented approach and its tool support integrates with our performance testing process using the *MBPeT* (Abbors et al., 2012) tool. The MBPeT tool generates load in real-time by executing the workload models in parallel. The parallel execution is meant to simulate the concurrent nature of normal web requests coming from real-world users. The tool itself has a distributed master/slave architecture which makes it is suitable for cloud environments. However, the approach proposed in this paper can be used independently for analyzing and classifying the usage of a web site.

The rest of the paper is structured as follow: In Section 2, we give an overview of the related work. In Section 3, we present the formalism behind the workload models that we use. In Section 4, we cover the methodology whereas, in Section 5, we present the tool support. Section 6 shows our approach applied

on a real-world example. In Section 7, we demonstrate the validity of our work. Finally, in Section 8, we present our conclusions and discuss future work.

2 RELATED WORK

Load testing is still often done manually by specifying load scripts that describe the user behavior in terms of a subprogram (Rudolf and Pirker, 2000), (Subraya and Subrahmanya, 2000). The subprogram is then run for each virtual user, possibly with the data being pre-generated or randomly generated. With regard to the data, theses types of approaches exhibit a certain degree of randomization. However, the behavior of each virtual user is a mainly a repetition of predefined traces. Most of these approaches are prone to errors due to much manual work and lack of abstraction that stochastic models offer. However, the question: "How to create a realistic stochastic performance model?" remains.

There exists a plethora of tools on the market that can analyze HTTP-based logs and provide the user with statistical information and graphs regarding the system. Some tools might even offer the user with common and reoccurring patterns. However, to the best of our knowledge, there is no web analytics software that will create a stochastic model from log data.

Kathuria et al. proposed an approach for clustering users into groups based on the intent of the web query or the search string (Kathuria et al., 2010). The authors divide the user intent into three categories: navigational, informational, and transactional. The proposed approach clusters web queries into one of the three categories based on a K-means algorithm. Our approach differs from this one in the sense that we cluster the users by their behavior by looking at the request pattern and accessed resources, whereas in their approach, the authors cluster users based on the intent or meaning behind the web query.

Vaarandi (Vaarandi, 2003) proposes a *Simple Logfile Clustering Tool* consequently called *SLCT*. SLCT uses a clustering algorithm that detects frequent patterns in system event logs. The event logs typically contain log data in various formats from a wide range of devices, such as printers, scanners, routers, etc. The tool automatically detects common patterns in the structure of the event log. The approach is using data mining and clustering techniques to detect normal and anomalous log file lines. The approach is different from ours in the sense that we assume that the logging format is known and we build a stochastic model that can be used for performance testing from common patterns found in the log.

Shi (Shi, 2009) presents an approach for clustering users interest in web pages using the K-means algorithm. The author uses fuzzy linguistic variables to describe the time duration that users spend on web pages. The final user classification is then done using the K-means algorithm based on the time the users spend on each page. This research is different from ours in the sense that we are not classifying users based on the amount of time they spend on a web page but rather on their access pattern.

The solutions proposed by Mannila et al. (Mannila et al., 1997) and Ma and Hellerstein (Ma and Hellerstein, 2001) are targeted towards discovering temporal patterns from event logs using data mining techniques and various association rules. Both approaches assume a common logging format. Although association rules algorithms are powerful in detecting temporal associations between events, they do not focus on user classification and workload modeling for performance testing.

Another approach is presented by Anastasiou and Knottenbelt (Anastasiou and Knottenbelt, 2013). Here, the authors propose a tool, *PEPPERCORN*, that will infer a performance model from a set of log files containing raw location tracking traces. From the data, the tool will automatically create a Petri Net Performance Model (PNPM). The resulting PNPM is used to make an analysis of the system performance, identify bottlenecks, and to compute end-to-end response times by simulating the model. The approach differs from our in the sense that it operates on different structured data and that the resulting Petri Net model is used for making a performance analysis of the system and not for load generation. In addition, we construct probabilistic time automata (PTA) model from which we later on generate synthetic load.

Lutteroth and Weber describe a performance testing process similar to ours (Lutteroth and Weber, 2008). Load is generated from a stochastic model represented by a form chart. The main differences between their and our approach is that we use different type of models and that we automatically infer our models from log data while they create the models manually. In addition, due to their nature, form chart models are less scalable compared to PTAs.

3 WORKLOAD MODELS

The work presented in this paper connects to our previous model-based performance testing process using the *MBPeT* (Abbors et al., 2012) tool. A workload model is the central element in this process, being used for distributed load generation. Pre-

viously, the model was created manually from the performance requirements of the system and based on an estimated user behavior. In order to model as realistic workload as possible, we use historic usage data extracted from web-server logs.

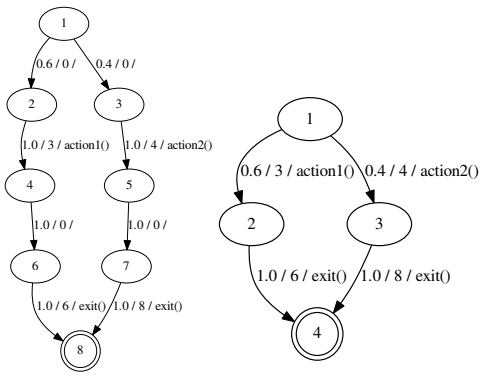
3.1 Workload models

Traditionally, performance testing starts first with identifying key performance scenarios, based on the idea that certain scenarios are more frequent than others or certain scenarios impact more on the performance of the system than other scenarios. A performance scenario is a sequence of actions performed by an identified group of users (Petriu and Shen, 2002). However, this has traditionally been a manual step in the performance testing process. Typically, the identified scenarios are put together in a model or subprogram and later executed to produce load that is sent to the system.

In our approach, we use *probabilistic timed automata* (PTA) (Jurdziński et al., 2009) to model the likelyhood of user actions. The PTA consists of a set of *locations* interconnected to each other via a set of *edges*. A PTA also includes the notion of time and probabilities (see Figure 1(a)). Edges are labeled with different values: *probability value*, *think time*, and *action*. The *probability value* represents the likelihood of that particular edge being taken based on a probability mass function. The *think time* describes the amount of time that a user thinks or waits between two consecutive actions. An *action* is a request or a set of requests that the user sends to the system. Executing an action means making a probabilistic choice, waiting for the specified think time, and executing the actual action. In order to reduce complexity of the PTA, we use a compact notation where the probability value, think time, and action are modeled on the same edge (see Figure 1(b)).

4 AUTOMATIC WORKLOAD MODEL CREATION

In this section, we describe the method for automatically creating the workload model from log data and we discuss relevant aspects in more detail. The starting point of our approach is a web server log provided by web servers such as Apache or Microsoft Server. A typical format for a server log is shown in Table 1. The log is processed in several steps and a workload model is produced.



(a) Original PTA (b) Compact PTA

Figure 1: Example of a probabilistic timed automata

4.1 Data Cleaning

Before we start parsing the log file we prepare and clean up the data. This entails that irrelevant data is removed from the log. Nowadays, it is not uncommon to encounter requests made by autonomous machines, also referred to as bots, usually used to crawl the web and index web sites. These types of requests are identified and removed from the log into a separate list.

At the moment, we are only interested in HTTP requests that result in a success or redirect (i.e., response codes that start with 2xx or 3xx). Requests that result in an error, typically response codes that start with 4xx or 5xx, are usually not part of the intended behavior and are also put in a separate list.

4.2 Parsing

The cleaned log file is parsed line by line using a pattern that matches the logging format. In our approach, a new virtual user is created when a new client IP-address[1] is encountered in the log. For each request made to the sever, the requested resource is stored in a list associated with a virtual user. The date and time information of the request together with the time difference to the previous request is also stored. The latter is what we denote as *think time* between two requests.

4.3 Pre-processing

From the previous step, we obtain a list of virtual users and for each virtual user a list of requests made

[1]Our approach uses IP-addresses for user classification since the UserId is only available for authenticated users and usually not present in the log.

Client IP-address	User-Identifier	User Id	Date	Method	Resource	Protocol	Status Code	Size of Object
87.153.57.43	example.site.com	bob	[20/Aug/2013:14:22:35 -0500]	GET	/browse	HTTP/1.0	200	855
87.153.57.43	example.site.com	bob	[20/Aug/2013:14:23:42 -0500]	GET	/basket/book/add	HTTP/1.0	200	685
87.153.57.43	example.site.com	bob	[20/Aug/2013:14:23:58 -0500]	GET	/basket/book/delete	HTTP/1.0	200	936
136.242.54.78	example.site.com	alice	[21/Aug/2013:23:44:45 -0700]	GET	/browse"	HTTP/1.0	200	855
136.242.54.78	example.site.com	alice	[21/Aug/2013:23:46:27 -0700]	GET	/basket/phone/add	HTTP/1.0	200	685
136.242.54.78	example.site.com	alice	[21/Aug/2013:23:57:02 -0700]	GET	/basket/view.html	HTTP/1.0	200	1772

Table 1: Requests to be structured in a tree

from the same client IP-address. In the pre-processing phase, these lists of requests are split up into shorter lists called *sessions*. A session is a sequence of requests made to the web server which represent the user activity in a certain time interval. It is not always trivial to say when one session ends and another begins, since the time interval varies from session to session. Traditionally, a session ends when a certain period of inactivity is detected, (e.g., 30 minutes). Hence, we define a session *timeout value* which is used to split the list of requests of a given user into sessions. In other words, we are searching for a time gap between two successive requests from the same virtual user that is greater than the specified timeout value. When a gap is found, the request trace is split into a new session. An example using a timeout of 30 minutes is shown in Figure 2.

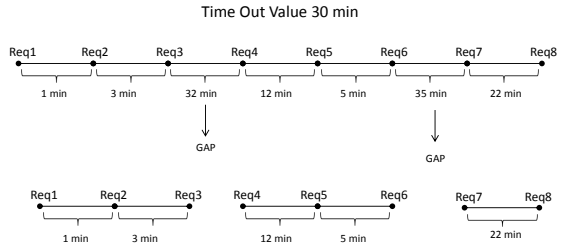


Figure 2: Example of splitting a list of requests into shorter sessions

4.4 Building a Request Tree

Visitors interact with web sites by carrying out *actions*. Actions can be seen as abstract transactions or templates that fit many different requests. These requests can be quite similar in structure, yet not identical to each other. For example, consider a normal web shop where users add products to the basket. Adding two different products to the basket will result in two different web requests even though the action is the same. In this step, we group similar requests into actions.

To achieve this, we first put the requests into a tree structure. For example, consider the example in Table 1. We split the string of the requested resource by the "/" separator and structure it into a tree. Figure 3-left shows how the requests in Table 1 would be structured. We always keep count of how many times we end up in a leaf node. For each new log line, we try to fit the request into the tree, otherwise a new branch is created.

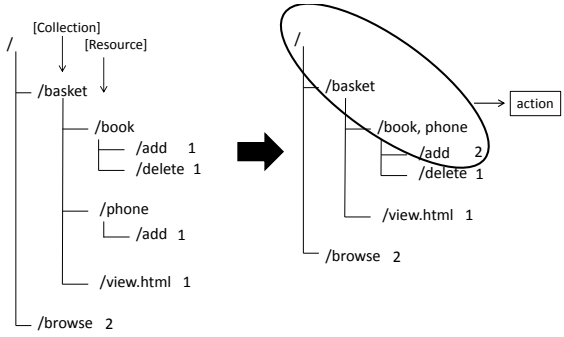


Figure 3: Example of request tree reduction.

After parsing a large log file, we obtain a large tree that might be difficult to manage. However, the tree can be reduced into a smaller tree by grouping together nodes. The algorithm is recursive and nodes at the same level in the tree are grouped together if they share joint sub-nodes. Figure 3-right shows how a tree can be reduced into a smaller tree. Once the request tree has been reduced as much as possible, every path in the reduced tree, that reaches a leaf node, is then considered as an *action* that can be executed against the system.

Consider the second request made by both Bob and Alice in Table 1. These two requests are basically the same type of request. They both request a resource from the same collection. This is similar to a *REST* interface where one uses *collections* and *resources*. It would seem obvious that these two requests are the result of the same action, only that the user requested different resources. Hence, by grouping together requests of the same type to the same resources, the tree can be reduced to a smaller tree. Similar requests are grouped into an *action*.

Requests in the tree can also be joined by manually inspecting the tree and grouping nodes that are a result of the same action. If a node in the path has more than one parameter, (e.g., it is a result of

grouping two resources) that part of the request can be parameterized. For example, the request "/basket/book,phone/add" is a parameterized action where either *book* or *phone* should be used when sending the actual request to the system.

4.5 User Classification

Before we start constructing a workload model representing the user behavior, we cluster different virtual users into groups according to their behavior. By user behavior we mean a behavioral pattern that a group of web site visitors have in common. A *User Type* can be seen as a group abstracting several visitors of a web site.

To group visitors based on the actions they perform we use the K-means algorithm (MacQueen, 1967). Table 2 shows the properties used for clustering. The properties are the *actions* obtained from the reduced request tree and the numbers represent the number of times a visitor has performed that action. Figure 5 show how the different visitors in Table 2 would be clustered into groups (or *User Types*) using the K-means algorithm. The only input in this step is the number of desired clusters which has to be specified a priori. Figure 4 depicts a typical example of clustering data into two groups using K-means. K-means clustering is an old method that involves assigning data points to k different clusters so that it minimizes the total squared distance between each point and its closest cluster center. One of the most widely used algorithms is simply referred to as "K-means" and it is a well documented algorithm that have several proposed optimization to it. The clustering is executed as follows:

1. Choose k clusters and initialize the centroid by uniformly choosing a random value from the data points.
2. For every sample in the data set, assign it to the closest cluster using the Euclidean distance.
3. Calculate a new centroid for every cluster by computing the average of all samples in the cluster.
4. Repeat step 2 and 3 until the clusters no longer change.

Virtual User	Action1	Action2	Action3	Action4	Action5
Visitor 1	2	0	0	3	3
Visitor 2	0	3	4	3	3
Visitor 3	1	0	1	8	9
Visitor 4	4	6	0	0	1
Visitor 5	0	0	4	8	7
Visitor 6	5	2	0	7	0

Table 2: Example showing the number of actions that different visitors perform.

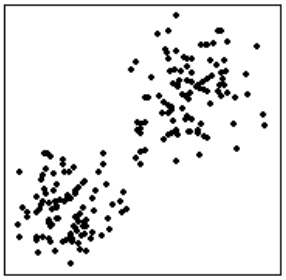


Figure 4: Example of two dimensional K-means clustering

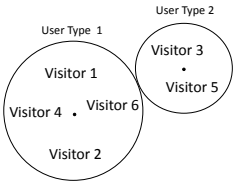


Figure 5: K-means clustering on data from Table 2.

Our approach also allows us to cluster virtual users based on other characteristics. Table 3 shows an example using different clustering parameters. Here the variable *#Get* means the total number of GET requests sent to the system and *#Post* means the total number of POST requests sent to the system. *ATT* stands for Average Think Time, *ASL* stands for Average Session Length, and *ADT* stands for Average Response Size.

Virtual User	#Get	#Post	ATT	ASL	ARS
Visitor 1	25	3	44	653	696
Visitor 2	17	0	25	277	1353
Visitor 3	31	3	54	1904	473
Visitor 4	19	1	23	444	943

Table 3: Example showing different clustering parameters.

This method, however, gives a different clustering result than the method presented previously and can be used as a complement if the first method gives an unsatisfactory result.

4.6 Removing Infrequent Sessions

Before we start building the workload model for each selected cluster, we filter out low frequency sessions. If we would include all possible sessions in the final workload model it would become too cluttered and difficult to understand and would include actions which do not contribute significantly to the load due to their low frequency rate. We are mainly interested in the common group behavior among visitors in the same cluster.

Removing sessions that have low frequency is achieved by sorting the sessions in descending order according to their execution rate. We filter out

low frequent sessions according to a Pareto probability density function (Arnold, 2008) by cutting off the tail beneath a certain threshold value. The threshold value is given as a percentage value. That means that sessions below the threshold are simply ignored and treated as irrelevant. The threshold value can however be adjusted on-the-fly to include more or fewer sessions in the workload model. Table 4 shows an example of sessions listed in a descending order according to their execution count. There is a total of 20 sessions, some of them have been executed several times, (e.g., session 1 has been executed 7 times). A threshold value of 0.7 would in this case mean that we want 70 percent of the most executed sessions included in our model, meaning a total of 14 sessions. Thus, we would have to construct a model with (Session1 * 7) + (Session2 * 6) + (Session3 * 1).

Session	Number of times executed
Session 1	7
Session 2	6
Session 3	3
Session 4	2
Session 5	1
Session 6	1
Total	20

Table 4: Sessions listed in a descending order according to number of times executed.

4.7 Building the Workload Model

The workload models that we create describes the common behavior of all virtual users belonging to the same cluster. We say that the model describes the behavior of a particular *User Type*. Creating the model for a particular user type is a step-wise process where we overlap sessions of all visitors belonging to the same cluster.

Session by session we gradually build a model, while reusing existing nodes in the model as much as possible. At each step, we note the number of times an edge has been traversed, the action, and the think time value. We use this information to calculate the probability and average think time of each edge in the model.

Figure 6 depicts how the workload model is gradually built. One session at a time is included in the workload model. An edge represents an action being sent to the system. The numbers associated to the edges represent session IDs. Each node represents a place, where the visitor waits before sending the next action. One by one we include all the session belonging to the same cluster, while reusing existing nodes as much as possible. Identical sessions will be laid on top of each other and at each step, we note the number of times an edge has been traversed, the action, and the think time value. We use this information to calculate the probability and average think time of each edge.

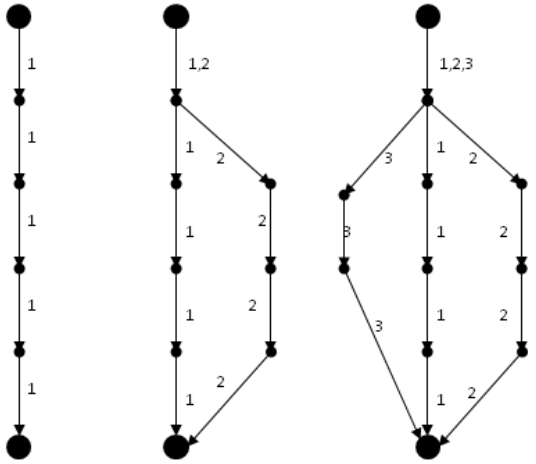


Figure 6: Model built in a step-wise manner.

We calculate the probability for an action as the ratio of a particular action to all the actions going out from a node. In a similar way, we calculate the think time of an action by computing the average of all think time values of an action.

In order to guarantee that the workload generated from the workload model matches the workload present in the log file, we calculate the user arrival rate. This information together with the distribution between user types is described in a higher level model called the *root model*. Figure 7 depicts such a model.

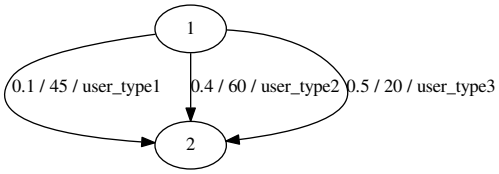


Figure 7: Root model describing different user types their waiting times and probability

The labels on the edges are separated by a "/" and refer to the *probability*, *waiting time*, and *user type*, respectively. The probability value describes the distribution between different user types. The waiting time describes the average waiting time between sessions. The user type value simply denotes what workload model to execute. To calculate the waiting time of a user type, we first have to study the waiting time between different sessions of a particular user. We then calculate the user waiting time by computing an

average time between sessions for every user belonging to a cluster.

5 TOOL SUPPORT

Tool support for our approach was implemented using the Python (Python, 2014) programming language. To increase the performance of the tool and make use of as many processor cores as possible for the most computation intensive tasks, we made use of Python's multiprocessing library.

Our tool has a set of pre-defined patterns for common logging formats that are typically used in modern web servers (e.g., Apache and Microsoft Server). However, if the pattern of the log file is not automatically recognized (e.g., due to a custom logging format) the user can manually specify a logging pattern via a regular expression. Once the log is parsed, the data is stored into a database. This way we avoid having to re-parse large log files from one experiment to another.

Before parsing a log file, the tool prompts the user for a session time out value and the number of user clusters. This information, however, has to be provided a priori. Once the file has been parsed and the reduced request tree has been built, the user has a chance to manually inspect the tree. Requests can be grouped manually by dragging one node on top of the other. Figure 8 shows an example of such a request tree.

When the workload models have been built for each cluster they are presented to the user. Figure 9 shows an example where 2 clusters have been used. The left pane shows the number of concurrent users detected throughout the logging period. The slider bar at the bottom of the figure can be used to adjust the threshold value, which determines how many sessions to include in the model. A higher threshold value usually means more sessions are included in the model, leading to a more complex model.

When saving the model, the tool will create two artifacts: the workload models and the Python adapter code. The latter contains the mapping of each action in the models in a parameterized form and is used to interface our MBPeT tool with the system under test.

6 Example

In this section, we apply our approach to a web log file containing real-users data.

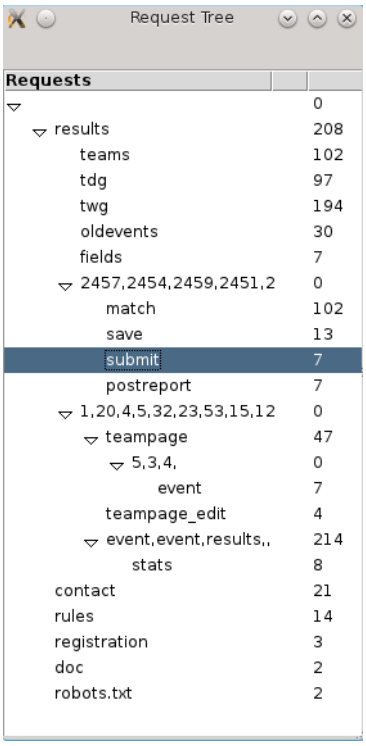


Figure 8: Example of the request tree

The web site[2] used in this example maintains scores of football games played in the football league called *pubiliiga*. It also stores information about where and when the games are played, rules, teams, etc. The web site has been created using the Django framework (Django Framework, 2012) and runs on top of an Apache web server.

6.1 Data Cleaning

The log that we used was 323 MB in size and contained roughly 1.3 million lines of log data. The web site was visited by 20,000 unique users that resulted in 365,000 page views between April 25th of 2009 and August 23rd of 2013. However, most of the users only visited the web site once or twice and there were only about 2,000 frequent users that regularly visited the web site. Also, since the web site is updated frequently on the same platform on which it is running, the log contained a significant amount of data from erroneous requests made by the simple method of trail and error during development. All erroneous requests and requests made from known robots were filtered out. The results that we are going to show in this section are generated from a selected section of the log data containing a mere 30,000 lines of log data, gen-

[2] www.pubiliiga.fi

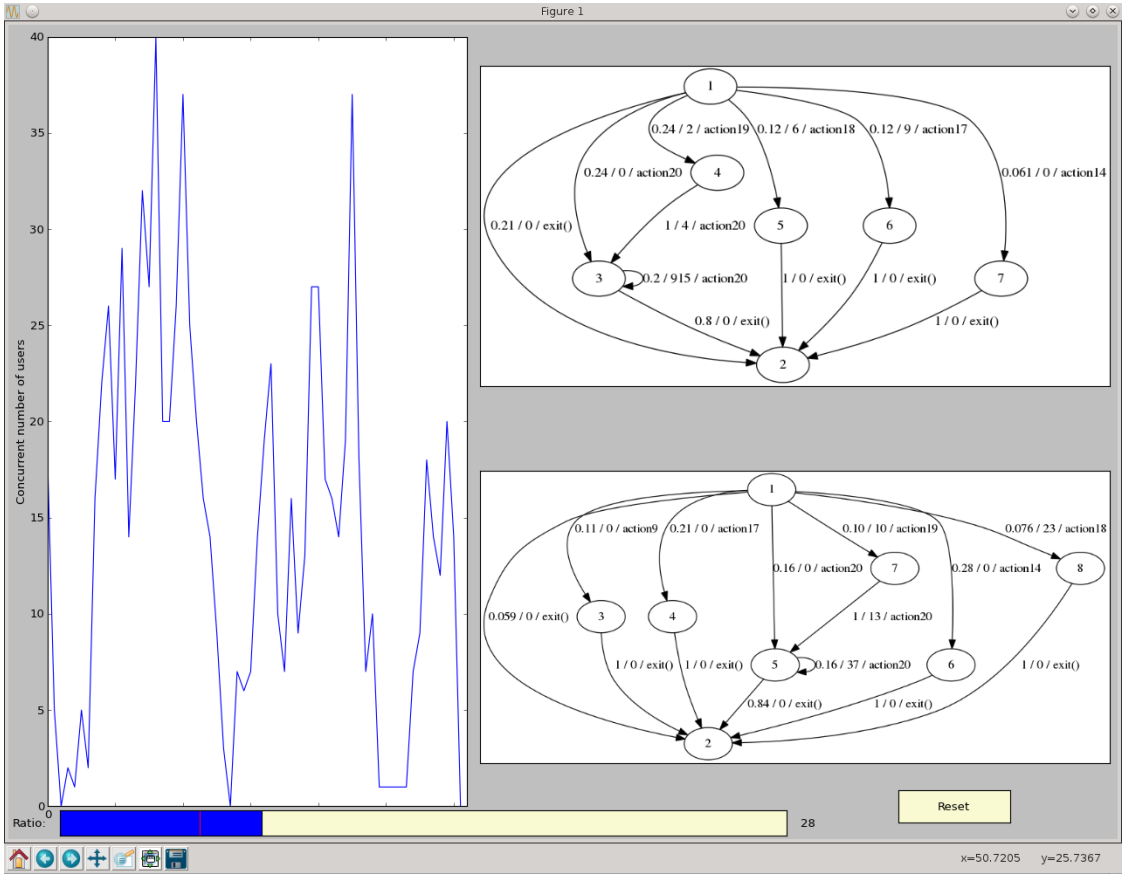


Figure 9: The output window

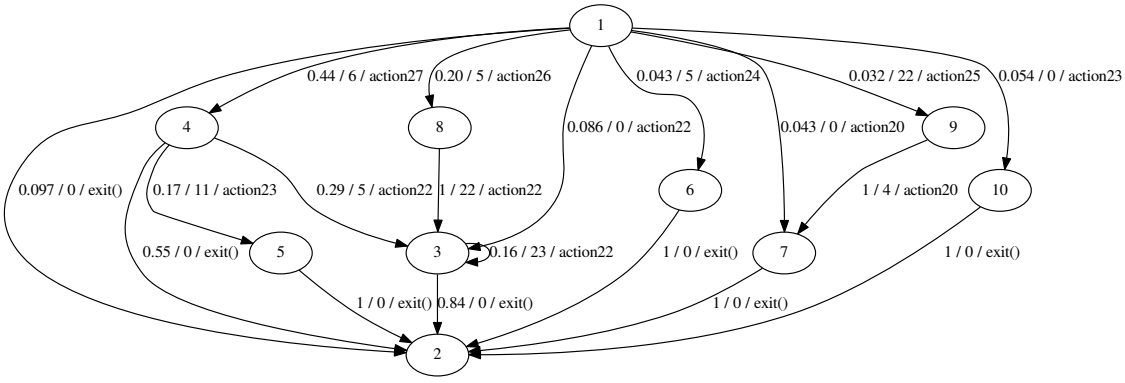


Figure 10: Workload models recreated from log data.

erated by 1092 unique users.

6.2 Pre-Processing

We used a session timeout value of 60 minutes to determine where to split the list of requests into sessions. In this experiment, we clustered the users into two different groups. The total time to parse and preprocess the data was around 10 seconds. The computer was equipped with a 8 core Intel i7 2.93 GHz processor and had 16 GB of memory.

6.3 Results

We used a threshold value of 0.3 when reconstructing the workload model for both clusters, meaning that 30 percent of most executed traces are included in the models. Figure 10 shows the workload model for cluster 1. A total of 985 virtual users were grouped into this cluster.

For confidentiality reasons the actual request types have been left out and replaced by abstract types. Creating the model took approximately 2 seconds. However, the execution time may hugely vary depending on the number of sessions that need to be included in the workload model. That number of sessions included in the model depends on what threshold value is selected.

7 VALIDATION

In this section, we demonstrate the validity of our approach on an auctioning web service, generically called YAAS (Yet Another Auction Site). The YAAS web service was developed as a university stand-alone project. The web service has a *RESTful* interface and has 5 simple actions:

- Browse: Returns all active auctions.
- Search: Returns auctions that matches the search query.
- Get_Auction: Returns information about a particular auction.
- Get_Bids: Returns all the bids made to a particular auction.
- Bid: Allows an authenticated user to place a bid on an auction.

During this experiment we preformed two load tests. First, we generated load from workload models that we built manually. We then re-created the workload models from the log data that was produced during first load test. In the second load test, load was generated from the re-created workload models. Finally, we compared the load that was generated during both tests. In the first step, we manually created models for two different user types. To test if the clustering works as expected, we made the workload models almost identical except for one request. One user type is doing distinctively a browse request while the other user type is always doing a search request. Figure 11(a) depicts the model for *user type 1*, the one that is performing distinctively a browse request. A similar model was also created for *user type 2*. If the algorithm can cluster users into different groups when only one action distinguishes them, then we consider the clustering to be good enough.

7.1 Generating a log file

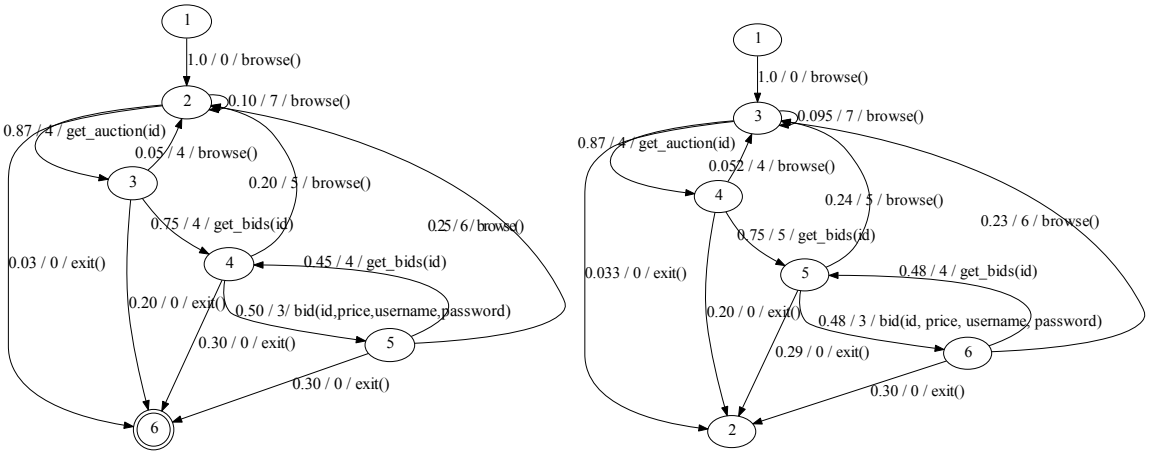
Once the models were built, they were used to load test the YAAS system using our in-house performance testing tool *MBPeT*. We simulated 10 virtual users (60% user type 1 and 40% user type 2) in parallel for 2 hours. We set the virtual users to wait 20 seconds between each session. This value is later going to influence the timeout value during pre-processing phase. From the produced log file, containing roughly 10,000 lines, we re-created the original models as accurately as possible. We point out that the original model is of a probabilistic nature, which means that distinctly different traces with different lengths can be generated from a fairly simple model. For example, the shortest session had only 1 action, while the longest session had 22 actions. Also, we do not have exact control over how many times each trace is executed by a user.

7.2 Recreating the models

To make sure that we split the sessions in a similar way we used a timeout value of 20 seconds. No other delay between the requests was that large. We also clustered the data into 2 user types. Each user type is later going to be represented with a separate workload model. In this experiment we did not filter out any user sessions, hence we used a threshold value of 1.0, meaning all traces found in the log were used to recreate the models. Figure 11(a) shows the original workload model while Figure 11(b) shows the reconstructed workload model for *User Type 1*. A similar model was also created for *User Type 2*. As one can see, the only difference from the original model is the probability values on the edges. However close, the probability values in the original models do not match exactly those in the generated workload models. This is due to the fact that we use a stochastic model for generating the load and we do not have an exact control of what traces are generated. Figure 12(a) shows original root model while Figure 12(a) shows the re-created root model. From the figures we can see that the probability values of the re-created root model match that of the original root model (60% and 40%) and that the waiting time is close to 20 seconds (19.97 and 19.98).

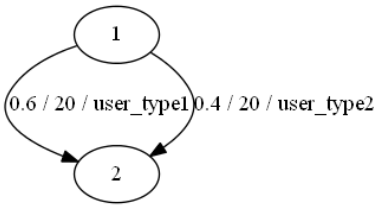
7.3 Data Gathering

Even though the models look similar, we also wanted to make sure that the load generated from the original models matched the load generated from our re-created models. Hence, we let the MBPeT tool mea-

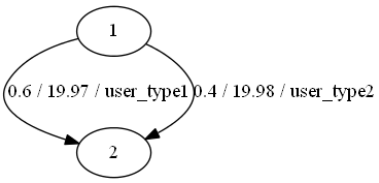


(a) User Type 1 original model. (b) User Type 1 recreated model.

Figure 11: Original and re-created workload models.



(a) Original root model.



(b) Recreated root model.

Figure 12: Root models

sure the number of requests sent to the YAAS system during both steps. Table 5 shows a comparison between the tests.

Request	Load Test 1	Load Test 2
Search(string)	1263	1294
Browse()	1895	1942
Get_Auction(id)	2762	2821
Get_bids(id)	2697	2625
Bid(id, price, username, password)	1288	1265
Total	9903	9947
Request Rate	1.37 req/sec	1.38 req/sec

Table 5: Comparison between the two test runs

As can be seen from the table, the re-created model produced a slightly higher workload. However, we like to point out that the load generation phase lasted for 2 hours and we see a difference of 44 requests. This is backed up by looking at the measured request rate. Load test 1 generated 1.37 req/sec, while load test 2 is virtually identical with 1.38 req/sec. Figure 13 shows the workload as a function of request (actions) over time for both test sessions. As one can see, the graphs are not identical but but the trend and scale is pretty much similar.

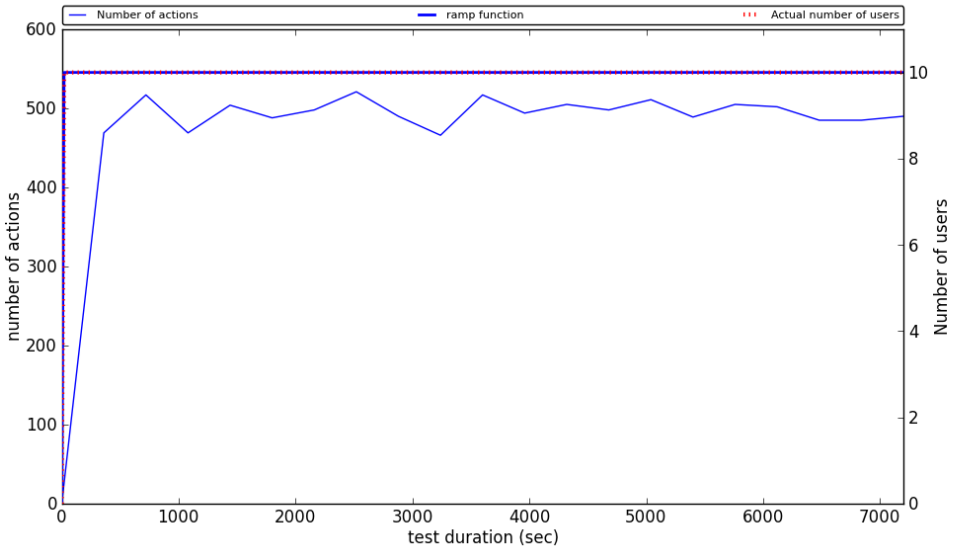
8 CONCLUSIONS

In this paper, we have presented a tool-supported approach for creating performance models from historical log data. The models are of a stochastic nature and specify the probabilistic distribution of actions that are executed against the system.

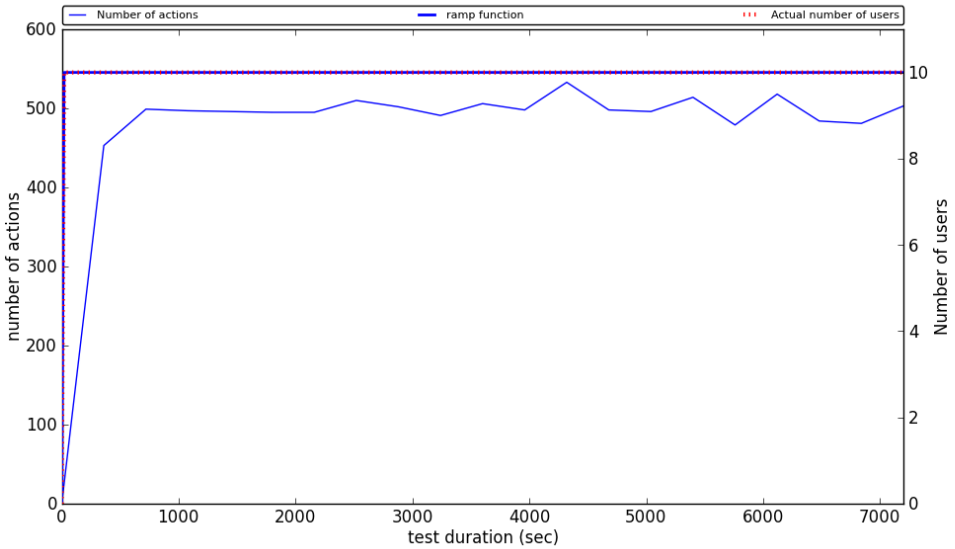
The approach is automated, hence reducing the effort necessary to create workload models for performance testing. In contrast, Cai et al. (Cai et al., 2007) report that they spent around 18 hours to manually create a test plan and the JMeter scripts for the reference Java PetStore application (Oracle, 2014).

The experiments presented in this paper have shown that the approach can adequately enough create workload models from log files and they mimic the real user behavior when used for load testing. Further, the models themselves give insight in how users behave. This information can be valuable for optimizing functions in the system and enforcing certain navigational patterns on the web site.

Future work will targeted towards handling larger amount of log data. Currently the tool is not optimized enough to operate efficiently on large data amounts. Another improvement is automatic session detection. Currently the tool follows a pre-defined timeout value for detecting sessions. Automatic session detection could suggest different timeout values



(a) Load Test 1



(b) Load Test 2

Figure 13: Number of request shows as a function over time

for different users, hence, improving on the overall quality of the recreated model. Currently, we are only clustering users according to accessed resources. In the future, we would like to extend the K-means clustering algorithm to cluster based on other relevant factors like: request method, size of resource, user request rate, etc. This clustering method could suggest models that, when executed, exercise the workload patterns on the system, thus, potentially finding ”hidden” bottlenecks. Further, an interesting experiment would be to analyze only failed or dropped requests. This way one could for instance study the details of how a DoS-attack was carried out and what pages were hit during the attack.

ACKNOWLEDGEMENTS

Our sincerest gratitude go to the owners of www.pubilliiga.fi for letting us use their data in our experiments.

REFERENCES

Abbors, F., Ahmad, T., Truscan, D., and Porres, I. (2012). MBPeT: A Model-Based Performance Testing Tool. *2012 Fourth International Conference on Advances in System Testing and Validation Lifecycle*.

Al-Jaar, R. (1991). Book review: The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling by raj jain (John Wiley & Sons). *SIGMETRICS Perform. Eval. Rev.*, 19(2):5–11.

Anastasiou, N. and Knottenbelt, W. (2013). Peppercorn: Inferring performance models from location tracking data. In *QEST*, Lecture Notes in Computer Science, pages 169–172. Springer.

Arnold, B. (2008). Pareto and generalized pareto distributions. In Chotikapanich, D., editor, *Modeling Income Distributions and Lorenz Curves*, volume 5 of *Economic Studies in Equality, Social Exclusion and Well-Being*, pages 119–145. Springer New York.

Cai, Y., Grundy, J., and Hosking, J. (2007). Synthesizing client load models for performance engineering via web crawling. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 353–362. ACM.

Django Framework (2012). Online at https://www.djangoproject.com/.

Ferrari, D. (1984). On the foundations of artificial workload design. In *Proceedings of the 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '84, pages 8–14, New York, NY, USA. ACM.

Jurdziński, M., Kwiatkowska, M., Norman, G., and Trivedi, A. (2009). Concavely-Priced Probabilistic Timed Automata. In Bravetti, M. and Zavattaro, G., editors, *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*, volume 5710 of *LNCS*, pages 415–430. Springer.

Kathuria, A., Jansen, B. J., Hafernik, C. T., and Spink, A. (2010). Classifying the user intent of web queries using k-means clustering. In *Internet Research*, number 5, pages 563–581. Emerald Group Publishing.

Lutteroth, C. and Weber, G. (2008). Modeling a realistic workload for performance testing. In *12th International Conference on Enterprise Distributed Object Computing.*, pages 149–158. IEEE Computer Society.

Ma, S. and Hellerstein, J. L. (2001). Mining partially periodic event patterns with unknown periods. In *Proceedings of the 17th International Conference on Data Engineering*, pages 205–214, Washington, DC, USA. IEEE Computer Society.

MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, number 1, pages 281–297. Berkeley, University of California Press.

Mannila, H., Toivonen, H., and Inkeri Verkamo, A. (1997). Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289.

Oracle (2014). Java Pet Store 2.0 reference application. http://www.oracle.com/technetwork/java/index-136650.html. Last Accessed: 2014-05-23.

Petriu, D. C. and Shen, H. (2002). Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications. pages 159–177. Springer-Verlag.

Python (2014). Python programming language. Online at http://www.python.org/. Last Accessed: 2014-05-23.

Richardson, L. and Ruby, S. (2007). *Restful web services*. O'Reilly, first edition.

Rudolf, A. and Pirker, R. (2000). E-Business Testing: User Perceptions and Performance Issues. In *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)*, APAQS '00, pages 315–, Washington, DC, USA. IEEE Computer Society.

Shi, P. (2009). An efficient approach for clustering web access patterns from web logs. In *International Journal of Advanced Science and Technology*, volume 5, pages 1–14. SERSC.

Subraya, B. M. and Subrahmanya, S. V. (2000). Object driven performance testing in web applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)*, pages 17–26. IEEE Computer Society.

Vaarandi, R. (2003). A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations and Management (IPOM03)*, pages 119–126. IEEE.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems
194. **Olli Lahdenoja**, Local Binary Patterns in Focal-Plane Processing – Analysis and Applications
195. **Annika H. Holmbom**, Visual Analytics for Behavioral and Niche Market Segmentation
196. **Sergey Ostroumov**, Agent-Based Management System for Many-Core Platforms: Rigorous Design and Efficient Implementation
197. **Espen Suenson**, How Computer Programmers Work – Understanding Software Development in Practise
198. **Tuomas Poikela**, Readout Architectures for Hybrid Pixel Detector Readout Chips
199. **Bogdan Iancu**, Quantitative Refinement of Reaction-Based Biomodels
200. **Ilkka Törmä**, Structural and Computational Existence Results for Multidimensional Subshifts
201. **Sebastian Okser**, Scalable Feature Selection Applications for Genome-Wide Association Studies of Complex Diseases
202. **Fredrik Abbors**, Model-Based Testing of Software Systems: Functionality and Performance

Turku Centre *for* Computer Science

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

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3247-3
ISSN 1239-1883

Fredrik Abbors

Model-Based Testing of Software Systems

Fredrik Abbors

Model-Based Testing of Software Systems

Fredrik Abbors

Model-Based Testing of Software Systems: Functionality and Performance