



Faculty of Economic Sciences, Communication and IT
Department of Computer Science

Johan Nordholm

Model-Based Testing: An Evaluation

Degree Project of 30 ECTS credit points
Master of Science in Information Technology

Date/Term: 2010-01-22
Supervisor: Donald F Ross
Examiner: Kerstin Andersson
Serial Number: E2010:01

Model-Based Testing: An Evaluation

Johan Nordholm

This thesis is submitted in partial fulfillment of the requirements for the Master's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Johan Nordholm

Approved, 2010-01-22

Advisor: Donald F Ross

Examiner: Kerstin Andersson

Abstract

Testing is a critical activity in the software development process in order to obtain systems of high quality. Tieto typically develops complex systems, which are currently tested through a large number of manually designed test cases. Recent development within software testing has resulted in methods and tools that can automate the test case design, the generation of test code and the test result evaluation based on a model of the system under test. This testing approach is called model-based testing (MBT).

This thesis is a feasibility study of the model-based testing concept and has been performed at the Tieto office in Karlstad. The feasibility study included the use and evaluation of the model-based testing tool Qtronic, developed by Conformiq, which automatically designs test cases given a model of the system under test as input. The experiments for the feasibility study were based on the incremental development of a test object, which was the client protocol module of a simplified model for an ATM (Automated Teller Machine) client-server system. The experiments were evaluated both individually and by comparison with the previous experiment since they were based on incremental development. For each experiment the different tasks in the process of testing using Qtronic were analyzed to document the experience gained as well as to identify strengths and weaknesses.

The project has shown the promise inherent in using a model-based testing approach. The application of model-based testing and the project results indicate that the approach should be further evaluated since experience will be crucial if the approach is to be adopted within Tieto's organization.

Acknowledgements

I would like to thank my supervisor Donald F Ross at Karlstad University for his guidance, feedback and encouragement during the writing of this thesis. I would also like to thank my supervisors Sören Torstensson and Robert Magnusson at Tieto for their support during the thesis project and for the instructive discussions about software testing as well as my project partner Yasir Malik, from Blekinge Institute of Technology, for a great team work. Finally, I would like to thank Athanasios Karapantelakis and Michael Lidén at Conformiq for the Qtronic course and all the support during the project.

Contents

- 1 Introduction 1**
 - 1.1 Motivation..... 2
 - 1.2 Goals 2
 - 1.3 Feasibility study 3
 - 1.4 Thesis Organization 4

- 2 Background..... 6**
 - 2.1 Introduction..... 6
 - 2.2 Software Development 6
 - 2.2.1 Process
 - 2.2.2 Requirements
 - 2.2.3 Specifications
 - 2.2.4 Testing
 - 2.3 Software Testing 9
 - 2.3.1 Testing levels
 - 2.3.2 Testing characteristics
 - 2.3.3 Test design techniques
 - 2.3.4 Summary
 - 2.4 Introduction to model-based testing 14
 - 2.4.1 Testing process
 - 2.4.2 Classic testing processes
 - 2.4.3 Testing at Tieto
 - 2.4.4 Summary
 - 2.5 Model-based testing 19
 - 2.5.1 What is model-based testing?
 - 2.5.2 Scope of MBT
 - 2.5.3 Approaches of MBT
 - 2.5.4 MBT process
 - 2.5.5 Benefits
 - 2.5.6 Limitations
 - 2.5.7 Summary
 - 2.6 Interfaces..... 27
 - 2.6.1 Test object (ATM)
 - 2.6.2 Qtronic
 - 2.6.3 Java
 - 2.6.4 TCL
 - 2.6.5 UML
 - 2.7 Summary..... 29

3	Experiment.....	31
3.1	Purpose	31
3.2	Test System.....	31
	3.2.1 Test tool: Qtronic	
	3.2.2 Test object	
	3.2.3 Test scripts	
	3.2.4 Test execution environment	
3.3	Introductory example.....	40
	3.3.1 Modeling	
	3.3.2 Test generation	
	3.3.3 Script rendering and test harness implementation	
	3.3.4 Test execution	
3.4	Design Factors in Modeling.....	47
3.5	Experiments	48
	3.5.1 Experiment 1	
	3.5.2 Experiment 2	
	3.5.3 Experiment 3	
	3.5.4 Experiment 4	
3.6	Summary.....	68
4	Results and Evaluation	69
4.1	Design Factors in Modeling.....	69
4.2	Experiment 1.....	70
	4.2.1 Model Development	
	4.2.2 Test generation	
	4.2.3 Test harness and test execution environment implementation	
	4.2.4 Test execution	
	4.2.5 Results and conclusions: experiment 1	
4.3	Experiment 2.....	76
	4.3.1 Model Development	
	4.3.2 Test generation	
	4.3.3 Test harness implementation	
	4.3.4 Test execution	
	4.3.5 Results and conclusions: experiment 2	
4.4	Experiment 3.....	83
	4.4.1 Model Development	
	4.4.2 Test generation	
	4.4.3 Test harness implementation	
	4.4.4 Test execution	
	4.4.5 Results and conclusions: experiment 3	
4.5	Experiment 4.....	88
	4.5.1 Model Development	
	4.5.2 Test generation	
	4.5.3 Test harness implementation	
	4.5.4 Test execution	
	4.5.5 Results and conclusions: experiment 4	
4.6	Summary of Experiment Results	93
4.7	Project Analysis	94
	4.7.1 Project work	
	4.7.2 Qtronic	
	4.7.3 Qtronic Modeler	

5	Conclusion.....	102
5.1	Results.....	102
5.2	Discussion.....	104
5.3	Future work.....	106
	5.3.1 Manual test design vs. MBT	
	5.3.2 Modeling of current applications at Tieto	
	5.3.3 Evaluation of the tool	
5.4	Final Comments.....	107
	References	108
A	Requirements and Specifications.....	110
A.1	Requirements	110
	A.1.1 User requirements	
	A.1.2 System requirements	
	A.1.3 Functional requirements	
	A.1.4 Non-functional requirements (quality requirements)	
A.2	Specifications.....	112
	A.2.1 Software requirements specification	
	A.2.2 System requirements specification	
	A.2.3 Software design specification	
	A.2.4 Component specification	
	A.2.5 Interface specification	
	A.2.6 Behavioral specification	
B	Testing	114
B.1	Terminology	114
	B.1.1 Glossary	
	B.1.2 Defects	
B.2	Testing techniques	116
	B.2.1 Static techniques	
	B.2.2 Black-box techniques	
	B.2.3 White-box testing techniques	
B.3	Traceability matrix.....	119
C	Test System	121
C.1	Specifications.....	121
	C.1.1 Version 1	
	C.1.2 Version 2	
	C.1.3 Version 3	
C.2	Qtronic	128
	C.2.1 Test design configuration parameters	
	C.2.2 Test generation options	
	C.2.3 Views for test generation result analysis	

List of Figures

Figure 1.1: Model-based testing approach	1
Figure 1.2: Feasibility study	4
Figure 2.1: Different kinds of testing	9
Figure 2.2: Overview of test design techniques [31]	12
Figure 2.3: Summary of different kinds of testing	13
Figure 2.4: Overview of the testing process at Tieto	16
Figure 2.5: Test phases at Tieto: actors and specifications	18
Figure 2.6: Scope of model-based testing [35]	20
Figure 2.7: The model-based testing process [35]	22
Figure 3.1: Qtronic stand-alone client	32
Figure 3.2: Status of the generation shown in Qtronic console	34
Figure 3.3: Test case interaction	35
Figure 3.4: ATM system overview	36
Figure 3.5: Qtronic example: Top-level state machine	42
Figure 3.6: Qtronic example: Authentication state machine	43
Figure 3.7: Qtronic example: Test Design Configuration	43
Figure 3.8: Qtronic example: Test generation progress	44
Figure 3.9: Qtronic example: Generated test cases	44
Figure 3.10: Qtronic example: Test execution	46
Figure 3.11: Experiment 1: Authentication state machine	50
Figure 3.12: Experiment 1: Authentication in top-level state machine	51
Figure 3.13: Experiment 1: State of top-level state machine	52
Figure 3.14: Experiment 2: Error handling	56
Figure 3.15: Experiment 2: QML transition example	57
Figure 3.16: Experiment 3: Biometric authentication	62
Figure 3.17: Experiment 4: QML model transition	64
Figure 4.1: Qtronic2 Client	97

Figure 4.2: Qtronic Modeler	100
Figure B.1 Example of a traceability matrix	119
Figure C.2: Specification 1: Top-level state machine	121
Figure C.3: Specification 1: Authentication state machine.....	122
Figure C.4: Specification 1: Withdrawal state machine.....	122
Figure C.5: Specification 2: Top-level state machine	123
Figure C.6 : Specification 2: Transfer state machine	124
Figure C.7: Specification 2: Deposit state machine	124
Figure C.8: Specification 3: Top-level state machine	125
Figure C.9: Specification 3: Biometric authentication state machine.....	126
Figure C.10: Specification 3: Balance state machine.....	126
Figure C.11: Specification 3: Withdrawal state machine.....	127
Figure C.12: Specification 3: Transfer state machine	127
Figure C.13: Specification 3: Deposit state machine	128

List of tables

Table 2.1: Overview of testing processes and approaches to test automation	27
Table 3.1: Results experiment 1	53
Table 3.2: Results experiment 2.....	59
Table 3.3: Results experiment 3.....	63
Table 3.4: Results experiment 4.....	67
Table 4.1: Results experiment 1	70
Table 4.2: Results experiment 1 and experiment 2	76
Table 4.3: Results experiment 2 and experiment 3	83
Table 4.4: Results experiment 3 and experiment 4.....	88
Table 4.5: Results summary	93

1 Introduction

The purpose of this thesis, and of the project undertaken for Tieto, is to evaluate the concept of model-based testing (MBT) and the model-based testing tool Qtronic. This thesis is a feasibility study of the MBT approach as well as a study of Qtronic. The feasibility study includes the development of a test object, the creation of a model and related tasks necessary to test the test object.

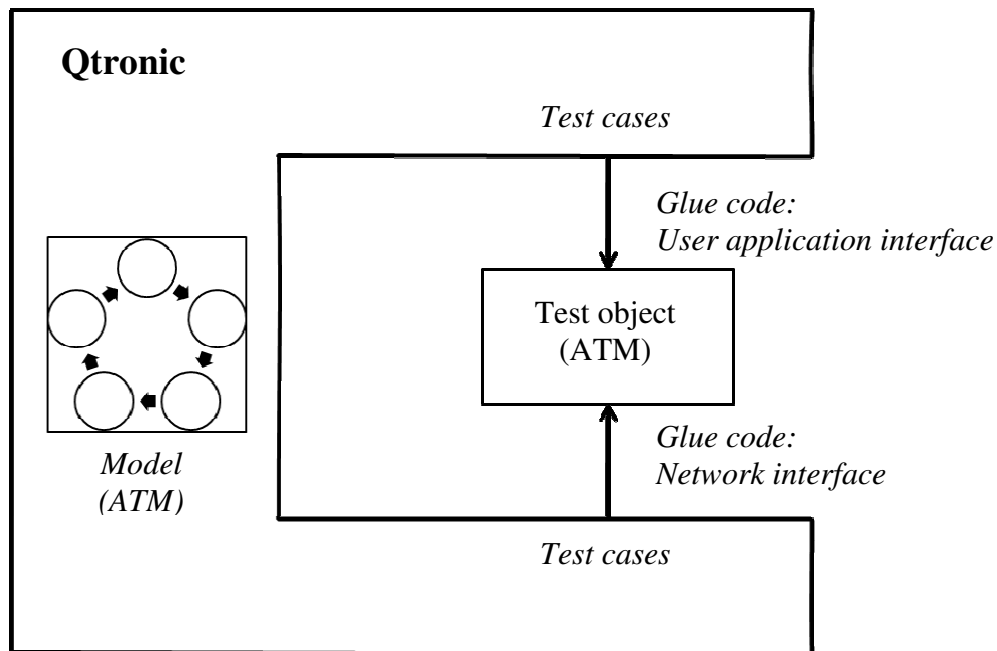


Figure 1.1: Model-based testing approach

Figure 1.1 above is an illustration of the thesis work. A test object is tested using a testing tool. The testing tool (Qtronic) uses a model, describing the outwardly observable behavior of the test object, to generate test cases automatically. To successfully execute the test cases against the test object, glue code, or a test harness implementation, is required. The glue code defines how the test scripts communicate with the system under test (the test object), i.e. sending input to and receiving output from the system under test.

MBT is a black-box testing technique where common testing tasks such as test generation and test result evaluation are automated based on a model of the system under test (SUT). This approach has recently spread to a variety of software domains but originates from hardware testing, most notably from telephone switches, and from the increasing use of object orientation and models in software design and software development [12]. When successfully deployed, it has been shown to yield economic benefits. Justifications of a software purchase or deployment of a change to an existing process include traditional metrics such as cost, quality and time to market. The methodology of MBT has proven its ability to provide improvements in all three of these areas [2].

1.1 Motivation

The Tieto office in Karlstad develops telecommunication products and provides testing services within the same area. Tieto is a supplier, or subcontractor, of telecommunication solutions and services for global actors, such as Ericsson. Thus projects at Tieto incorporate complex systems and products. In the development process of large and complex systems testing is a critical activity in order to obtain a system of high quality. Testing must be done continuously during the development of a system, as its functionality is gradually increasing, and is necessary on a daily or at least weekly basis. Systems developed at Tieto typically have a large number of states as well as a large number of input and output signals. Thus the number of test cases is very high. Normally it is not possible to test all combinations of state transitions so a representative subset of the important use cases has to be tested.

Currently much of the software testing at Tieto makes use of a script-based testing approach, where the test scripts are written manually by testers. For complex systems this approach results in a large number of test cases which have to be manually designed and maintained as the system evolves. Furthermore, the test scripts are dependent on the tester implementing the tests, hence test scripts may be difficult to maintain. Recent development within software testing has led to the development of methods and tools that can generate test code automatically, based on a model of the system to be tested. This testing approach is called MBT.

The MBT approach is a black-box testing technique which automates testing tasks such as test generation and test result evaluation [12] and has when deployed shown to yield economic benefits [2]. Moreover, reports claim that software defects can be found earlier in the development process using a MBT approach compared to the use of manual testing practices [5]. The use of models to depict the behavior of a system is a proven and major advantage in software development [11] as well as in software testing [2]. Software models are now accepted as a part of modern object orientated analysis and design. Modeling is a good way of capturing knowledge about a system and then reusing this information as the system grows. The model may also be used as a means of communication between different teams in an organization during development and the model may be reviewed by new team members to quickly come up to speed. For test teams, models provide a useful mechanism for structured analysis of the system [2].

1.2 Goals

The general goal of this thesis is to evaluate the concept of MBT. This general goal includes a pre-study of the concept as well as a feasibility study. The initial goals of the feasibility study are to learn the MBT tool Qtronic and to implement a test object. The general goal of the feasibility study is to apply and evaluate MBT, to document experiences and to make recommendations for future work. To summarize, the purpose of the feasibility study is to prove the concept but also to produce guidelines for future reference, as if MBT were to be adopted at Tieto.

The general goal of the thesis work was further divided into sub-goals.

1. Learn the concepts of model-based testing
2. Learn to use a tool (Qtronic) for modeling, test code generation and test execution
3. Develop a framework for a finite state machine and implement a logic on the framework (with incrementally increased complexity) to be used as a test object.
4. Develop a model for the test object in the test tool and try different criteria for generation of test code
5. Establish the test environment including development of “glue” between the test code and the test object.
6. Execute the tests and incrementally make the test object and the model more complex.
7. Evaluate the result, document the experience gained and make recommendations.

The project goals listed above will be performed as they are listed, with the exception that all goals except the initial goal, to study the theory behind the MBT concept, will be performed in an iterative and incremental fashion.

1.3 Feasibility study

The general purpose of the feasibility study is to evaluate the concept of MBT as well as a specific MBT tool, namely Qtronic. Specifically, the purpose is to develop a test object and to test this test object using Qtronic. The idea is that the test object will be developed incrementally while documenting experiences and results from the testing process. The test object, and consequently the model, will be extended and modified for different experiments. Each experiment will focus on different aspects for evaluation. Since the test object will be incrementally developed the results and the experiences of the experiments will be compared.

The test object is a part of a simplified model for an ATM client-server system, using an application layer protocol. The complete system specification includes the ATM (the client) and the central bank computer (the server), which communicates over a network. Both the client and the server are divided into subsystems. The client consists of the user interface, where the user can withdraw money and request account balance information, and a protocol module. The server has a corresponding structure. The test object in the feasibility study is limited to the protocol module of the client, which consists of a finite state machine which handles incoming messages from two interfaces and responds with outgoing interfaces for both interfaces. Thus the finite state machine describes the state of the protocol module and which outgoing message that follows an incoming message. The specifications for the finite state machines are defined in Appendix C.1.

Qtronic is the MBT tool that will be used. Qtronic automatically designs test cases given a model of the system under test (SUT) as input. The generated test cases are black-box tests, which mean that they evaluate the SUT based only on the system’s external behavior. Thus the models do not need to reflect the test object implementation structurally given that they describe the intended outwardly observable behavior [8]. The design models are expressed in the Qtronic Modeling Language (QML), and its graphical notation, which is defined in a separate tool, namely Qtronic Modeler. The generated test cases in Qtronic are abstract test cases that then are rendered to an executable format. In this thesis the test cases will be rendered to TCL scripts.

Qtronic renders the test cases to executable test scripts. Glue code, or a test harness, is then implemented using a test execution environment, i.e. a means of communicating with the SUT. The test harness is necessary to execute the test cases against the SUT.

Feasibility study

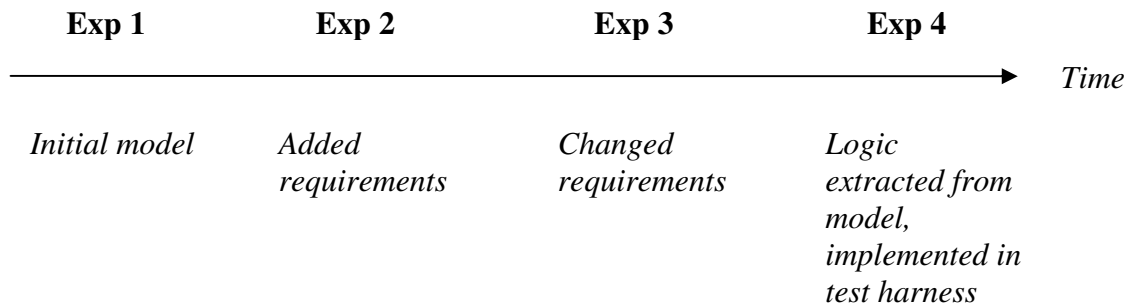


Figure 1.2: Feasibility study

As Figure 1.2 indicates, this thesis includes four experiments. The approach of the thesis work is to incrementally develop the test object and the model for each experiment as if MBT were deployed in a new project within the organization. Since the experiments are based on incremental development of the test object each experiment is analyzed individually but will also be compared to the previous experiment, except for the initial experiment. The purpose of the initial experiment is to create a working model of the test object and to successfully execute the generated test cases. The purpose of the second experiment is to extend the specification, thus also the test object and the model, to evaluate how added requirements propagate through the process. The purpose of the third experiment is to change the authentication requirements for the protocol module to see how changed requirements propagate through the process. The specification of this experiment will require a successful biometric authentication to complete account requests against the central bank computer. A second goal of the third experiment is to model a reusable structure, i.e. encapsulate logic that can be reused in multiple state machines in the model. The purpose of the fourth experiment is to extract some logic from the model and implement this logic in the test harness, while using the same version of the specification and testing the same version of the SUT as for the third experiment. For the specifications see Appendix C.1.

1.4 Thesis Organization

This thesis consists of five chapters. In chapter 2 software testing is discussed in general before introducing the concept of MBT. Software testing is discussed to place MBT in the perspective of traditional testing approaches and its testing scope. Chapter 3 includes a description of the whole test system, an introductory example of the testing process in this project as well as experiment descriptions and corresponding results. The analysis and the discussion of each experiment are provided in chapter 4. Chapter 4 also includes an analysis of this project and evaluations of Qtronic and Qtronic Modeler. In chapter 5 the conclusions

from the project are described. The conclusions include the results, a discussion of the project and suggestions for future work that may be performed in the context of this project area.

2 Background

2.1 Introduction

This background chapter includes the pre-study of the project. The chapter starts with section 2.2 by briefly discussing the process of software development and the roles of requirements and specifications in the process. Section 2.3 is an introduction to software testing, including characteristics and levels of testing as well as testing techniques. The concept of model-based testing (MBT) is introduced in section 2.5 and includes the different approaches, the process as well as benefits and limitations. The chapter ends with section 2.6 which briefly describes the interfaces used in this project.

2.2 Software Development

As an introduction to software testing some basic concepts of software development will be discussed. These will be described to place software testing in an overall perspective of the software development process as well as to introduce the requirements and basic ideas for testing.

2.2.1 Process

A software process is a set of activities and associated results which lead to the production of a software product. Such activities may include software development from scratch or by extending and modifying existing systems [33].

Today there are a number of software development processes deployed in the industry, such as Scrum [32], eXtreme Programming [3] and Rational Unified Process [23]. However, there is no ideal process and different organizations have developed different approaches to software development. Although there are many different software processes used, there are fundamental activities which are common to all software processes [33]:

1. *Software specification*: The functionality of the software and constraints on its operation must be defined.
2. *Software design and implementation*: The software to meet the specification must be produced.
3. *Software validation*: The software produced must be validated to ensure that it fulfills its purpose.
4. *Software evolution*: In many cases the software must evolve to meet the changing customer requirements.

2.2.2 Requirements

Software engineers often have to solve complex problems. Understanding the nature of the problems can be very difficult, especially if the system is new, and hence it is difficult to establish exactly what the system should do. The descriptions of the services and the constraints are the requirements for the system [33].

A software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate [1].

Since requirements may come from different levels of an organization it is useful to distinguish between these levels of description. One way to make this distinction is by using the terms user requirements and system requirements [33]. Furthermore, system requirements are often classified as functional or non-functional requirements [33]. Non-functional requirements are also sometimes referred to as constraints or quality requirements [1]. These different types of requirements are described in Appendix A.1.

2.2.3 Specifications

Specifications take all the information stated in customer requirements as well as any unstated but mandatory requirements and define what the product will be, what it will do, and how it will look [26]

Software systems typically have a large number of requirements, and the emphasis is shared between performing numerical quantification and managing the complexity of interaction among the large number of requirements. In software engineering a specification typically refers to the production of a document which can be systematically reviewed, evaluated, and approved [1].

Software specifications may be developed at three different levels: user requirements, system requirements and a software design specifications. The user requirements specification is the most abstract specification and the software design specification the most detailed. The added system requirements specification is somewhere in between, since it does not include implementation details but is detailed enough to describe system properties and constraints precisely [33].

Different levels of specification are useful because they communicate information about the system to different types of readers. User requirements should primarily be written for client and contractor managers, while system requirements should target senior technical staff and project managers. Finally, software design specifications should be written for the software engineers developing the system [33].

For a more detailed discussion regarding software specifications, see Appendix A.2.

2.2.4 Testing

IEEE [18] defines testing as “the process of analyzing a software item to detect differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.”

Testing itself is a process related to two other processes called verification and validation [6]. IEEE [18] defines the combined concept of verification and validation as “the process of determining whether requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements.”

Thus, referring to section 2.2.1 above, software validation is really a subset of software testing. Furthermore, IEEE [18] defines verification and validation separately. Verification is defined as “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [18].” Validation is defined as “the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [18].”

Verification is the process of evaluating whether or not the software being developed meets the requirements and the specification. An example of verification is to confirm that a model, developed during design, satisfies the requirements specified in an earlier development phase. In contrast to verification, customer participation is mandatory for validation since the primary focus of validation is customer satisfaction. Customer participation may be a useful support in verification as well, but is not required. Validation is the process in which customers and developers examine if the development results in the product the customer desires. The goal is to ensure and prove that the solution is adequate regarding the customer’s requirements [20].

To summarize, verification could be described as the process of confirming that software meets its specification, whereas validation could be described as the process of confirming that it meets the user’s requirements. One should never assume that the specification is correct [26].

Specifications and requirements are essential for testing. If there is no specification or requirements to compare against, there can be no testing. Without a specification it is hard to state that a system behavior is invalid, unless one has a convincing argument or a high professional credibility. It may be hard to convince developers that the behavior really is wrong. Further, it is virtually impossible to automate testing if there is no system specification or system requirements for the expected response. An automated test program cannot make on-the-fly subjective judgments about the correctness of the outcome [17].

2.3 Software Testing

There are various different kinds of testing. One way to illustrate and classify these kinds is along three dimensions, as in Figure 2.1 below [35].

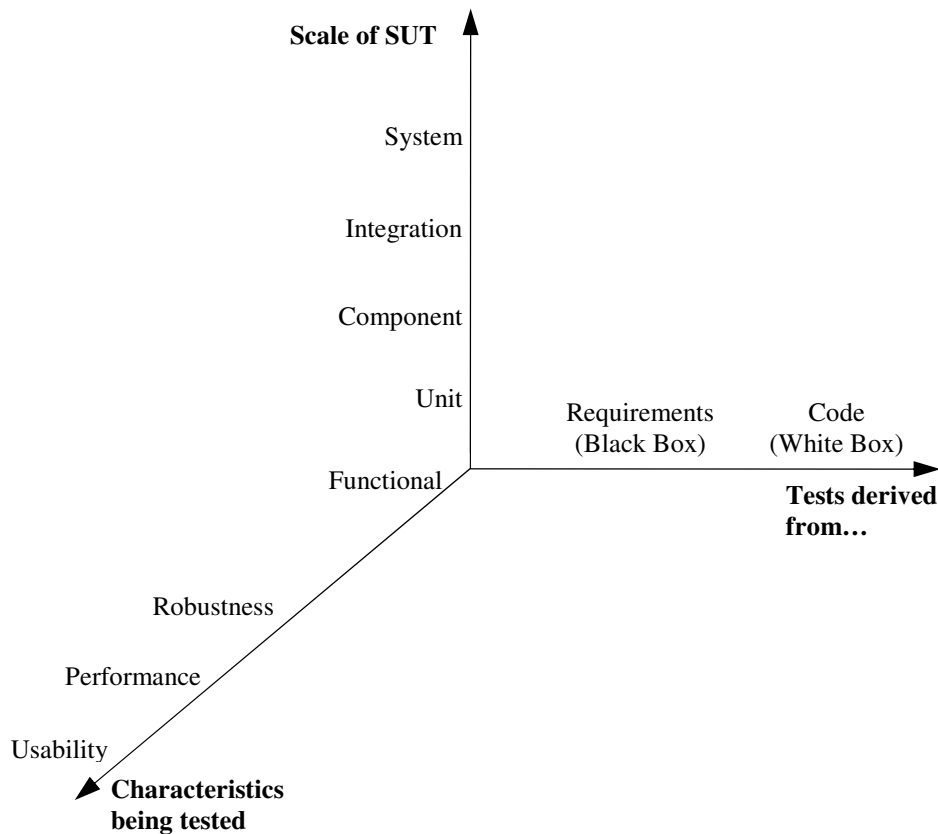


Figure 2.1: Different kinds of testing

One axis shows the scale of the system under test (SUT), ranging from units up to the whole system. The horizontal axis shows what kind of information is used to design the tests. The third axis shows different characteristics that can be tested, including functional (behavioral testing), robustness testing, performance testing and usability testing. These various kinds of testing are discussed in the remaining of this section.

2.3.1 Testing levels

Software can be tested at many different testing levels [6][26][31][33][35], for example at levels of units, components, of integration and of the whole system [35]. Testing levels are really levels of abstraction in terms of what is to be tested, and in what detail [6].

Different authors define different levels of testing in different ways and not always consistently. Not all terms appear in each collection of definitions. For example integration tests may be subdivided into different categories. The testing levels are related to the scale of the system under test (SUT), ranging from small components to the entire system.

Unit testing

Unit testing tests a single unit at a time [35]. A unit is the smallest piece of software subjected to testing, typically a class in object-orientated software development, and for example a procedure in C [9]. Unit testing is intended for checking the smallest testable parts, is of technical nature and require technical knowledge about the programming language. Thus, unit testing is often carried out as part of the developers' coding work. The basis for unit testing is software design specifications (see Appendix A.2). Suitable techniques include control flow testing and data flow testing [31].

Component testing

Component testing tests each component, or subsystem, separately [35]. Traditionally stubs have been used at this testing level as a white-box testing technique to test components. Stubs in the context of software testing receive or respond to data that the system under test sends [26].

Sometimes there is confusion regarding the difference between unit and component testing [26][31][35]. They may even be regarded as synonymous [31]. However, unit testing is most often performed by software developers during coding work [31], and involves demonstrating that an individual software unit has been implemented correctly [6]. Ryber [31] defines a component as a collection of code which realizes a function.

However, in this context components are really subsystems which potentially could be of significant complexity [35]. Subsystems have the characteristic that they can operate as independent systems in their own right [33].

Inputs to component testing may include a project test plan, system requirements (functional requirements), component specifications (see Appendix A.2.4) and the component implementation. The latter provides the information necessary to construct white-box and interaction test cases [25]. Component testing is performed by the developers since they provide the necessary information for testing at this level [33].

Integration testing

When individual components have been tested, they are integrated to a subsystem or to a complete system. When the system is constructed it is important to test the resulting system for problems that arise from component interactions, such as interface misuse of some component. Integration of system components may lead to complex interactions between the components and when anomalous output is detected it may be difficult to find the source of the error. Moreover, integration testing aims to discover a particular type of error namely interface errors. These cannot be discovered by testing individual components, since such errors are a result of the interaction between components and not the isolated behavior of a single component. Implementation of system features may spread across a number of components. Repairing errors may be difficult because it affects the whole group of components that implement a system feature [33].

Integration testing should be based on a written system specification (see Appendix A.2). This specification can be a detailed system requirements specification or a user-oriented specification of the features that should be implemented in the system. Integration testing is usually performed by an independent testing team [33].

The difference between component and integration testing is unclear for object-oriented systems, since objects are the basic structure used at all stages in the object-oriented development process [33].

System testing

While component testing and integration testing are performed iteratively, system testing aims to test the entire product, or at least a major part of it, at once [26]. System testing may be defined as the test executed by the developer that should demonstrate that the developed system or subsystems meet the functional and quality requirements (see Appendix A.1). That is, system testing should verify that all system requirements are met [27]. Both functional and non-functional requirements are included, and at this stage tests focus on external functionality rather than the internal software structure [31]. System testing focuses on errors at the highest level of integration [9] and is usually performed by an independent testing team [33].

2.3.2 Testing characteristics

As well as breaking down the testing process into test levels the testing process may also be divided into different types of characteristics being tested.

Functional testing

Functional tests are normally conducted from the user interface [17]. Functional testing (also known as behavioral testing) aims to find errors in the functionality of the system, for example testing that the correct outputs are produced for given inputs [35]. Functional tests at the system level are used to ensure that the behavior corresponds to the software requirements specification (see Appendix A.2). Furthermore, functional tests are normally black-box tests [6].

Robustness testing

Robustness testing aims to finding errors in the system under invalid conditions, such as unexpected inputs, unavailability of dependent applications, and hardware and network failures. This testing characteristic is used to verify that the system can cope with circumstances that are not expected [35]. Robustness may be defined as the degree to which the information system reacts as intended even after an interruption [27].

Performance testing

The goal of performance testing is to verify that the software meets the performance requirements specified in terms of functional requirements and quality requirements (see Appendix A.1) [6]. Performance testing (sometimes referred to as load testing) is testing the load-bearing ability of a system. One example could be to verify that the system can process the required number of transactions for a given time period [17], or to try how many simultaneous connections an Internet server can handle [26]. This kind of testing tests the limits of the system, for example if the software operates on peripherals such as printers or communication ports, connect as many as you can [26]. These tests are usually run to determine how quickly the system runs, in order to device whether optimization is needed [21].

Usability testing

Usability testing concerns user interface problems, that might make the software difficult to use or making users misinterpret the system output [35]. “Usability is a quality factor that is related to the effort needed to learn, operate, prepare input, and interpret the output of a computer program [6].”

Usability can be divided into the following quality factors: understandability, learnability, operability, attractiveness and usability compliance [27] (see Appendix A.1.4 for description of these quality factors).

2.3.3 Test design techniques

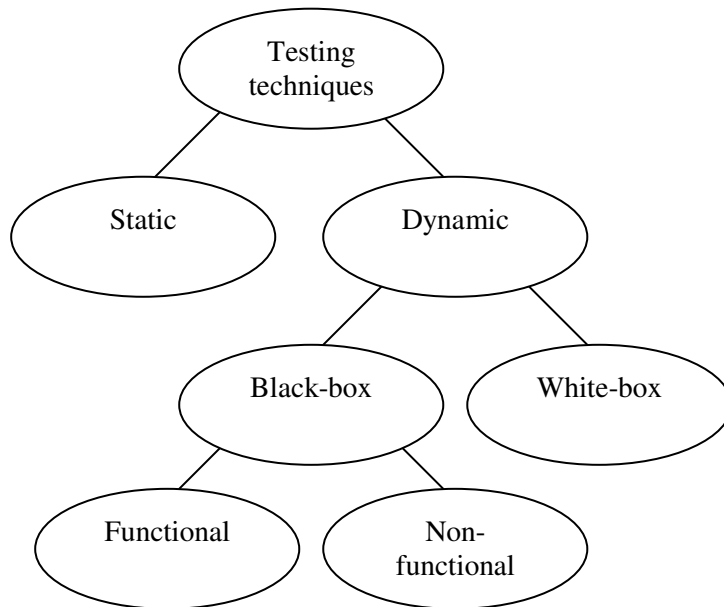


Figure 2.2: Overview of test design techniques [31]

Static techniques

A static test design technique does not involve program code being executed. Using a static technique involves different types of documentation in the form of text, models or code that are analyzed, often by hand. The defects found are often related to requirements and design, and typical examples of static techniques are inspections, walkthroughs, technical reviews and informal reviews [31] (see Appendix B.2.1 for definitions of these four static techniques).

Dynamic techniques

Dynamic testing techniques involve testing code by execution. These techniques can be divided into black-box (behavior-based) and white-box (structural) techniques [31], as shown in Figure 2.2.

Black-box testing

Black-box testing is a classification of test design techniques that derive the test cases from the externally visible properties of an object without having knowledge of the internal structure of this object. Using these techniques, the system is viewed as it would be in actual use [27]. Black-box testing techniques are based on functional requirements and quality requirements (4), and could thus be classified into two types: functional (using functional requirements) and non-functional (using quality requirements) black-box testing techniques [31].

Functional black-box testing

The description of the behavior and functionality of the system under test comes from functional requirements and specifications. The tester provides the specified inputs to the system under test, runs the test and then determines whether the output corresponded to those in the specification. This is what is typically referred to as black-box testing, and alternative methods are equivalence class partitioning, boundary value analysis and state transition testing [17].

Non-functional black-box testing

Non-functional black-box testing makes use of quality requirements. These quality requirements may further be categorized as functionality, reliability, usability, efficiency, maintainability and portability, where each category could be divided into subcategories. These characteristics are important to consider when testing software [31] (see Appendix A.1.4 for definitions of these quality requirements).

White-box testing

White-box testing is a technique that derives test cases from the internal properties of an object, with the knowledge of the internal set-up of the object [27].

The tester selects test cases to exercise the internal structural elements to determine if they behave as they are intended to. Typical methods used for white-box testing are statement testing, branch testing, path testing, mutation testing and loop testing [6] (see Appendix B.2.3 for definitions of these techniques).

2.3.4 Summary

This section described different kinds of testing in terms of testing levels, testing characteristics and testing techniques.

The different kinds of testing described in this section are summarized in Figure 2.3.

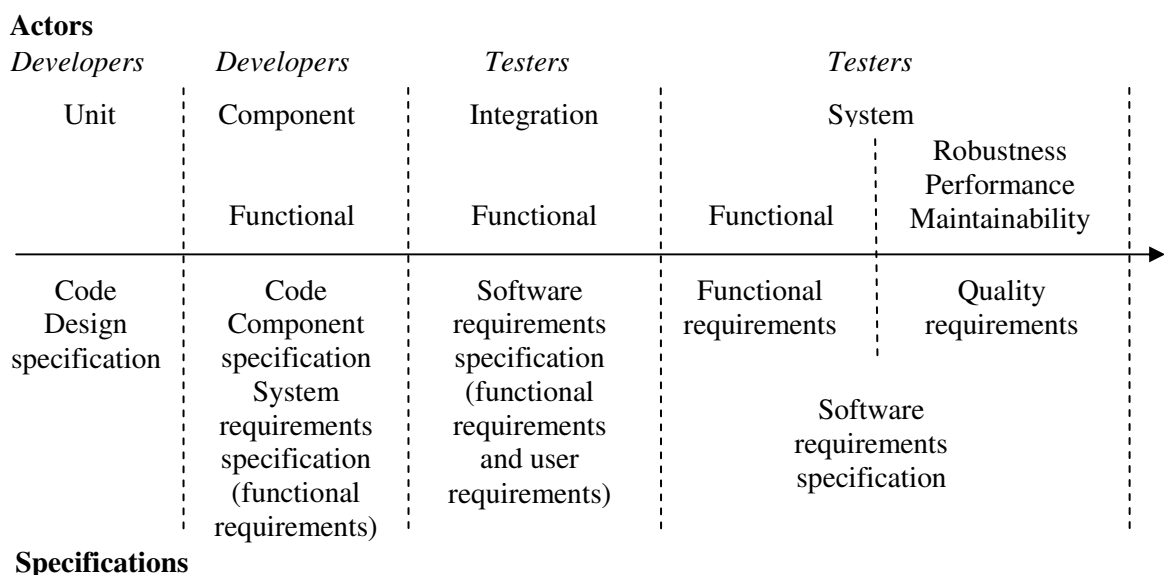


Figure 2.3: Summary of different kinds of testing

The test design techniques are not included since they cannot generally be mapped to actors and the specifications described in this thesis.

To summarize, unit and component testing make use of design specifications and code. At these levels of testing the abstraction level is relatively low, and white-box testing techniques are often used, and testing is performed by the developers. Component testing however may also make use of functional requirements and functional testing, typically black-box testing.

Integration testing is most often performed by an independent test team, even though the difference between integration and component testing is not really clear in the object-oriented development process. Integration testing is typically based on functional system requirements or user-oriented requirements.

System testing involves testing the whole system, including both functional and quality system requirements, as well as higher-level requirements that may have been defined. At the system level really all software characteristics are tested.

2.4 Introduction to model-based testing

In this section classic testing processes will be described, as a prelude to model-based testing (MBT), which is an alternative to these processes. This section also describe how testing is exercised at Tieto.

2.4.1 Testing process

Functional testing consists of three key issues [35]:

Designing the test cases

First, the test cases have to be designed, using system requirements while also considering test objectives and policies. Each test case is defined by a test context, a scenario and some evaluation criteria [35]. It is also a good idea to perform modeling before actually writing the test cases, since tests will primarily be based on the tester's mental model of the system [21].

Executing the tests and analyzing the results

The test cases then have to be executed on the system under test. Test outputs and results are then analyzed to evaluate the system behavior, and possibly to determine the cause of each test execution failure [35].

Verifying how the tests cover the requirements

It is vital to measure in which way the requirements are covered by the test suites in order to manage the quality of the testing process, and therefore also the quality of the product. One common way to do this is to use a traceability matrix (see Appendix B.3 for an example). A traceability matrix shows the link between functional requirements and test cases, generally in form of a many-to-many relation since one requirement may be covered by several test cases [35].

2.4.2 Classic testing processes

As an introduction to MBT some classic testing processes will be briefly described in this section.

A manual testing process

The test design is performed manually, using informal requirements, and the output is a human readable document that describes the desired test cases. However, manual test design is time-consuming. Also, manual design does not guarantee systematic coverage of the system functionality [35].

Test execution is also performed manually, by hand. The tester performs the specified steps of the test case by manually interacting with the SUT, compares the system output and analyzes the results. For some applications it is not possible to interact with the SUT, and in those cases a test execution environment may be used to interact with the system. However, execution is still performed manually. The manual test execution process is repeated for every new release of the SUT that needs to be tested [35].

A capture/replay testing process

This testing process attempts to reduce the cost of test re-execution by capturing the interactions with the SUT during one test execution and then replaying those interactions during later test execution. To capture and replay the interactions a tool is used. Test cases are still performed manually however. When a new release needs to be tested the tool can rerun all the tests and report which ones have failed [35].

The main problem is that just a small change in the interface may cause a large number of tests to fail, due to a lack of abstraction in the recorded tests. Hence, the key issue of automating the execution is only partially solved [35].

A script-based testing process

This approach solves the test execution problem by automating it, using test scripts. Test scripts are executable and run one or more test cases. Initializing the SUT, putting the SUT in the required context, creating test input values, passing those values to the SUT, recording the responses from the SUT, comparing those against expected outputs, and finally assigning a pass/fail verdict to each test, are all typically performed by a test script [35].

The problem with this approach is the maintenance issue. The test scripts not only have to be modified when requirements change, but also when implementation details change (for example some parameter). Maintenance for the test scripts can become very costly, and abstraction really is the key to reducing those costs [35].

A keyword-driven testing process

The goal of this approach is to overcome the maintenance issues of low-level test scripts by raising the abstraction level of test cases. The idea is to express each test case as abstractly as possible, but still precise enough to be executed and interpreted by a test execution tool. This is done by using action keywords in the test cases, along with data. Each action keyword corresponds to a piece of a test script, This allows the test execution tool to translate a sequence of keywords and data values into executable tests [35].

The higher abstraction level reduces the maintenance problems because test cases can often be adapted to a new version of the SUT or its environment easily, by updating the test scripts associated with a few keywords. However, test data and oracles are still manually designed [35]. A test oracle is a document or piece of software that allows testers to determine whether a test has been passed or failed [6].

Given these four testing processes, it can be concluded that they all share a common denominator, namely manual test design [35].

2.4.3 Testing at Tieto

This section will describe the general testing process at Tieto as well as describe the test phases in more detail, including actors and resources for each test phase.

Testing process

Designing as well as testing is performed incrementally at Tieto. The overall testing process is described in Figure 2.4 below.

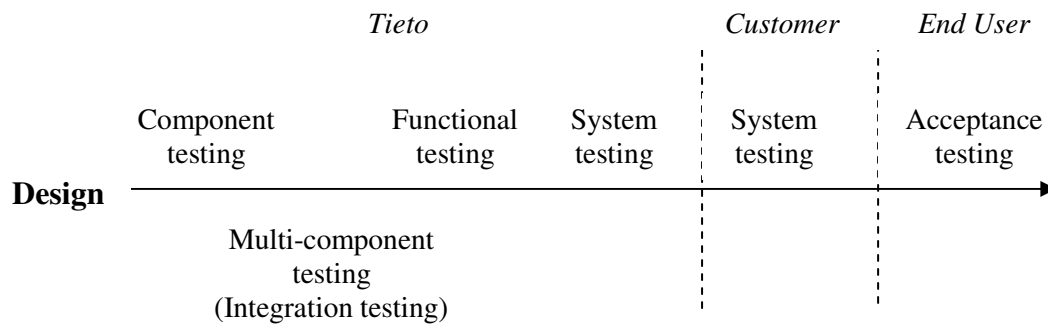


Figure 2.4: Overview of the testing process at Tieto

The process described in Figure 2.4 above seems to be linear, but really development and testing is carried out in an incremental and iterative fashion at Tieto, meaning that component, multi-component, functional testing and system testing are performed iteratively as systems are developed incrementally, i.e. the process is repeated for every delivery or shipment. Multi-component testing may or may not be performed, depending on the system being developed. Multi-component testing integrates multiple components to evaluate and verify that they work together, and is really integration testing (as described in section 2.3.1).

Software is often released in versions and delivered continuously to the customer. Before a delivery Tieto performs system testing (see section 2.3.1) to verify the system. When the customer receives a delivery in the development phase they also perform system testing to make sure that no major errors are encountered. Deliveries to the customer are more frequent than releases to the end user. Deliveries to the end user are called shipments, and are more crucial since it is here that the system will be used.

Acceptance testing is performed by the end user, and is the means by which customers approve what has been delivered. In simple terms, if the system does not solve the problems it was built to solve, development has not been successful [31]. The reason for Tieto's customer not performing acceptance tests is that Tieto is a consultant company and most often works together with their customers on projects, in a joint effort. It is then the Tieto's customer that sells and delivers products to their customers (end-users). It is thus the end-user who performs the acceptance tests.

Figure 2.4 does not mention unit testing. The reason is that this is really a part of the developers' duties as software units are developed. Furthermore, functional testing is the only testing characteristic included in Figure 2.4. Usability testing is not included in the figure of the simple reason that it is not performed at Tieto. The reason for this is that usability testing

often includes detailed testing of user interfaces, but the systems developed at Tieto are often embedded systems and user interface testing is not necessary.

However, robustness, performance and maintainability are all characteristics tested at Tieto. These characteristics are included in the system testing activity. Robustness, performance and maintainability testing are performed before shipments, that is, deliveries to the end-user. This is because shipments are more crucial than deliveries to the customer, since the system will be used by the end-user.

Maintainability is a quality characteristic of software and may be described as the ease with which the information system can adapt to new demands from the user, to changing external environments, or in order to correct errors [27] (see Appendix A.1.4 for a description of maintainability). Since many systems developed at Tieto are sophisticated systems that may take years to develop, and to do so incrementally, maintainability is valuable software characteristic.

Performance and robustness are important characteristics to test because they are often specified in terms of quality requirements. Also, testing these characteristics includes measurements that are useful to compare the capabilities of the system between different shipments.

One more piece is missing in Figure 2.4, namely *regression testing*. Regression testing is not a testing level. It is the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new errors have been introduced due to the changes. Regression testing may occur at any level of test [6]. Since regression testing is meant to test whether the system as a whole still functions, it will be executed frequently. New releases of a system often involve minor changes and the system functionality remains largely the same. Thus, regression test cases are very reusable and require in general only minor adaptations for each new system release [27].

The testing process at Tieto could be described as a script-based testing process (see section 2.4.2). They use their own automation framework, built by test scripts developed in TCL (see section 2.6.4). This framework contains libraries of test scripts.

Different kinds of testing

This section relates the previous discussed theory of requirements, specifications and different types of testing to the software testing performed at Tieto. Software testing at Tieto is really a subset of the general software testing theory.

This is illustrated in Figure 2.5 below.

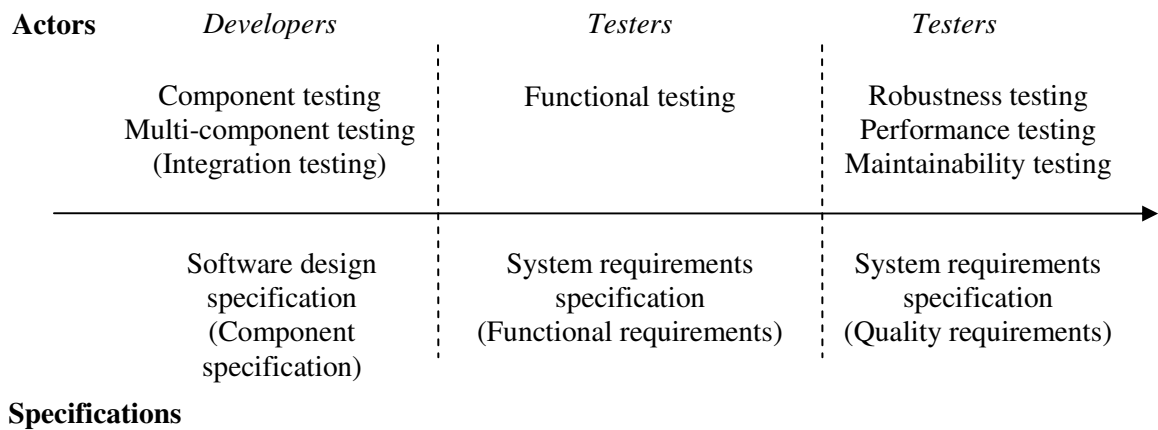


Figure 2.5: Test phases at Tieto: actors and specifications

Compared to Figure 2.4 this figure includes the actors and the resources used to perform different kinds of testing. Although Tieto uses specific name conventions for different types of specifications and documents this is what it translates to, from a higher level of abstraction.

Component and multi-component testing are performed by developers since development of new software components often make use of existing ones. Thus it is the developers' responsibility to test that components work as intended. Also, component testing often make use of stubs as a white-box testing technique and the primary input is a design specification, thus developers are suited to perform these tests.

Functional testing is performed by testers since the functional testing normally is black-box testing. This test phase make use of funtional requirements and do not require knowledge about the internal structure of the SUT. Functional requirements often refer to acceptable mappings between system input values and the corresponding system output values.

The third test phase in Figure 2.5 includes testing of software characteristics. This test phase could really be considered system testing and is similar to functional testing, but with the difference that quality requirements (or non-functional requirements) are used. Hence the system is tested under special conditions for performance, robustness and maintainability testing. During maintainability testing at Tieto maintenance activities by the end-user are simulated, such as adding a new signaling point, thus involving executing the system.

2.4.4 Summary

This section gave an introduction to MBT by first describing the general testing process, including the creation of test cases, the analysis of the test case execution results and the verification of how functional requirements are covered by test cases.

Furthermore, to put MBT in a perspective of the software testing evolution, existing and classical testing processes were discussed. The discussions included descriptions of the processes, but also brief descriptions of which testing problems they solve and do not solve.

Finally, this section also described the current testing process at Tieto. Since Tieto is interested in MBT and would like to evaluate the concept, as well as a specific tool, it is important to re-examine the existing testing process and methodology.

2.5 Model-based testing

This section will describe the concept of model-based testing (MBT), starting with a background and introduction. The scope of MBT, different approaches, the process as well as benefits and limitations will also be presented.

2.5.1 What is model-based testing?

During the last decade there has been a growth in black-box testing techniques. These are collectively called MBT. MBT is a general term that signifies an approach where common testing tasks such as test generation and test result evaluation are based on a model of the system under test. This approach has recently spread to a variety of software domains but originates from hardware testing; most notably telephone switches [12].

MBT can be considered the latest generation of test automation. Features of this concept include defect prevention (see Appendix B.1.2), early requirement defect detection (see Appendix B.1.2) and automated test generation from models, which eliminates manual test design and hence reduces cost. The main advantage of this technique is that by automatically generating tests using models of system requirements and specified functionality both the test design and the test execution process can be automated. By applying MBT, defects can be found earlier in the development process compared to the use of manual testing practices [5]. From a process perspective it also promotes more continuous testing activities, thus having organizational impacts [5] and supporting incremental development [2].

The key concept in this approach is the model, which therefore should be discussed. Models are hardly a new concept, but they are used within many disciplines to understand, specify and develop systems. Software models are now accepted as part of modern object oriented analysis and design. Modeling is a good way of capturing knowledge about a system and then reusing this knowledge as the system grows. Moreover it can be used as a means of communication between different teams in an organization during development. For test teams, models provide a mechanism for structured analysis of the system. However, the greatest benefit of models is in reuse since the work done is not lost. The next test cycle can start where the current one left off. New product features can be added to the model, tests extended and potentially new team members can quickly come up to speed by reviewing the model [2].

MBT has originated from the increasing use of object orientation and models (including UML) in software design and software development [12].

To summarize, the use of models to depict the behavior of a system is a proven and major advantage in software development [11] as well as in software testing [2]. Models can be utilized in different ways during the product life-cycle, including: improved quality of specifications, code generation, reliability analysis and test generation [2].

MBT offers many advantages, such as a high degree of automation, ability to generate high volumes of non-repetitive (unique) useful tests, means to evaluate regression test suites and the possibility of estimating a number of statistical measures of software quality [11]. However, not all areas of application have enjoyed successful application of MBT. The technique has proven its worth within areas such as embedded systems, user interfaces, and state-oriented systems. However, since the approach is new and not widely deployed no conclusions can be made that it suits testing of all types of applications [12].

Nevertheless, when successfully deployed it has shown economic benefits. Justifications of a software purchase or deployment of a change to an existing process include traditional metrics: cost, quality and time to market. The methodology of MBT has proven its ability to provide improvements in all three of these metrics [2].

2.5.2 Scope of MBT

When is MBT really applicable and how does it relate to the different kinds of testing (see section 2.3)?

The scope of MBT in relation to the various kinds of testing may be described as in Figure 2.6 (extension of Figure 2.1).

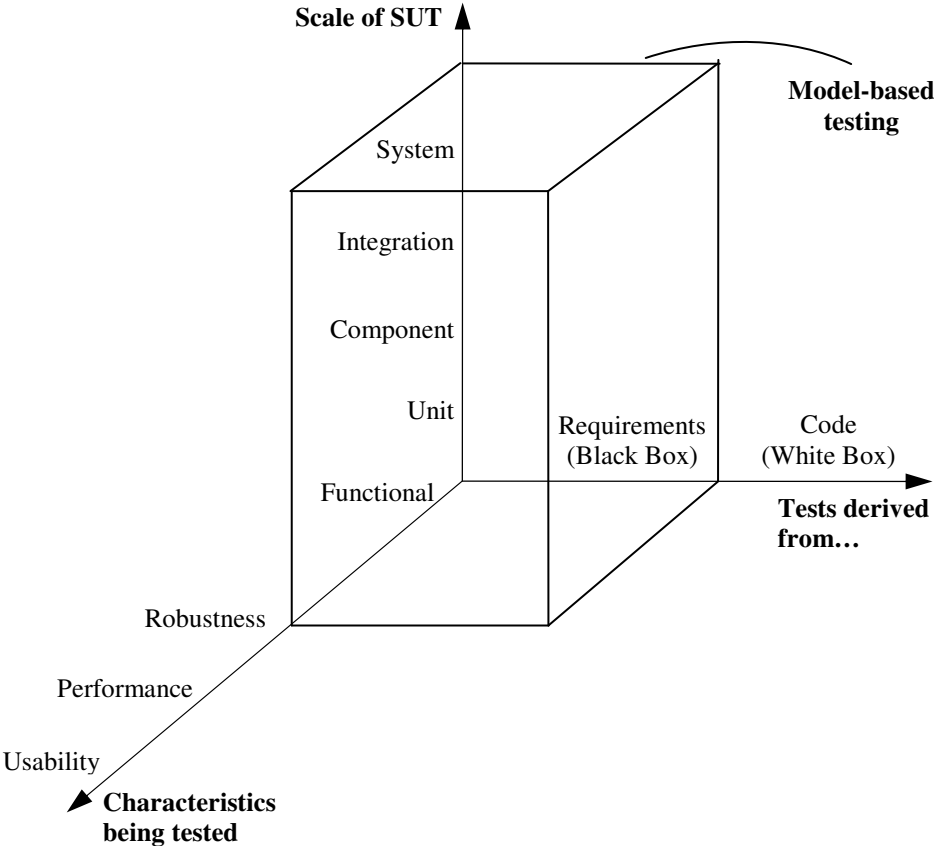


Figure 2.6: Scope of model-based testing [35]

As the diagram indicates, MBT is a black-box testing technique. In reality MBT is the automation of black-box test design [34]. MBT is in theory applicable to all levels of testing of the SUT, however for large systems the degree of complexity involved in creating a model may be unmanageable. For highly complex systems it may not be feasible to apply this methodology because of modeling overhead. Moreover black-box testing is usually used as a functional testing technique. The latter is often referred to as behavioral testing and aims at finding errors in the functionality of the system, which is specified in the model [35].

The main use of MBT is the generation of functional test cases, but it can also be applied for some kinds of robustness testing such as testing the system with invalid inputs. It is not yet widely applied for performance testing, but this is an area under development [35].

2.5.3 Approaches of MBT

The term MBT is currently used for a wide variety of test generation techniques. The four main approaches known as MBT are described by Utting and Leguard [35]:

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

These approaches will be addressed briefly below.

Generation of test input data from a domain model

In this approach the model provides the information about the domain of the input values. The test generation involves clever selection and combinations of a subset of those values to produce test input data. This approach is advantageous from a practical point of view, but it does not solve the complete test design problem because it cannot provide any test verdict [35].

Generation of test cases from an environment model

This approach uses a model to describe the environment of the system under test (SUT). Sequences of calls to the SUT can be generated from this model, but generated calls do not specify the expected output of the SUT. The environment model does not cover the behavior of the system, meaning that it is not possible predict the output values. In other words it is difficult to determine whether a given test passed or failed [35].

Generation of test cases with oracles from a behavioral model

The third meaning of MBT is the generation of executable test cases which include oracle information or some automated check on the actual output values to see if they are correct. Oracle information consists of input values associated with operations and the corresponding expected output values. This is a more challenging task than the two previously mentioned approaches. The test generator must know enough about the expected behavior of the SUT, such as the relationship between input and output, in order to generate test cases with oracles. Hence, the model must describe the behavior of the SUT. This is the only approach of the four that addresses the whole test design problem from choosing input values and generating sequence calls to generating executable test cases that include oracle information [35].

Generation of test scripts from abstract tests

The final approach assumes an abstract description of a test case, such as a UML sequence diagram, and focuses on transforming that abstract test case into a low-level test script that is executable. The model is the information about the structure and API (Application Programming Interface) of the SUT, and the details of how to transform a high-level call into executable test scripts [35].

2.5.4 MBT process

MBT automates the detailed design of test cases and the generation of the traceability matrix, which measures the coverage of requirements for each test case. Instead of writing hundreds of test cases, the test designer constructs an abstract model of the system under test. The MBT tool is used to generate a set of test cases from that model. As well as having the advantage of reducing design time, a variety of test suites can also be generated from the same model simply by using different test selection criteria [35]. The MBT process can be described as in Figure 2.7 below.

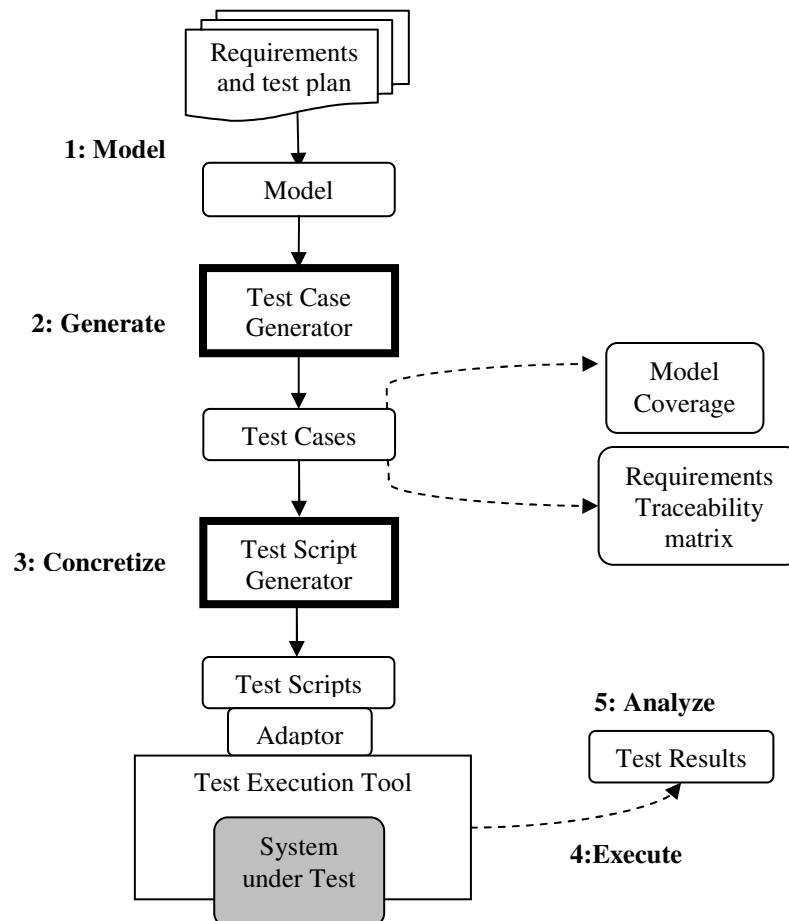


Figure 2.7: The model-based testing process [35]

The process can be portioned into five main steps:

1: Model the SUT and/or its environment

The first step of MBT is to construct an abstract model; really a simplified (behavioral) model of the system being tested. It should not be very detailed, but it should focus on the key aspects to be tested and be based on the specified requirements [35].

2: Generate abstract test from the model

After creating the behavioral model of the system the next step is to generate abstract test cases from that model. To do so, a test selection criteria needs to be specified, since the number of possible tests may be infinite. For example, interaction with the test generation tool

might be necessary to focus on a particular part of the model or to choose a particular model coverage criterion, such as to cover all transitions in a finite state machine [35].

Since the model is an abstraction of the SUT the abstract test cases are not directly executable. A requirements traceability matrix (see Appendix B.3 for an example), which links functional requirements with test cases to determine which requirements are covered by an individual test case, or various coverage reports are additional outputs of this step for most MBT tools. Coverage reports indicate how well the test cases cover the behavior of the model, while a requirements traceability matrix traces the link between functional requirements and generated test cases [35].

3: Concretize the abstract tests to make them executable

When the abstract test cases are generated they need to be transformed into executable concrete tests. This may be done by some separate transformation tool or it could be done by writing some adaptor code that implements each abstract operation to map against the lower-level SUT interface. The goal of this step is to connect the abstract test cases with the concrete SUT by adding details not included in the abstract model [35].

This two-layer approach has the advantage of being independent of the language used to write tests and of the test environment. Simply by changing the adaptor code, the tests can be reused in different test execution environments [35].

4: Execute the tests on the SUT and assign verdicts

Given the executable test scripts it is time to execute them against the SUT. Using online MBT, the tests will be executed as they are produced. In this case the MBT tool handles execution and recording of the results [35].

With offline MBT, a set of concrete test scripts has been produced. Hence the existing test execution tools and practices can be used [35].

5: Analyze the test results

The final step is to analyze the results of the test execution, but also to take the correct action given the findings. For each failed test it must be determined what caused the failure. It might be due to a fault in the SUT or to a fault in the test case itself. In the latter case this must be due to a fault in the adaptor code or in the behavioral model of the system. Hence, feedback about the correctness of the model is given in this step [35].

2.5.5 Benefits

Model checking can ensure that properties, like consistency, are not violated. Models also help refining unclear and poorly defined requirements. In MBT tests are generated to verify the SUT behavior as the models are refined. In this way model defects can be eliminated before coding begins. Other advantages of a model-based approach include automation of test case design, and generation of test scripts resulting in a more efficient testing process, significant cost savings, and in the end higher quality code [5].

Some of the benefits of using MBT are discussed briefly below.

Nature of the Model

The model used for test generation can either be a functional model of the system under test, or of the environment of the system, or a model of both. Models of the system are useful for the outputs of a system, which allows test oracles to be generated, and environment models

are useful for focusing the test generation on an expected usage of the system [34]. Modeling is a precise communication tool among teams in an organization. Certain models are sometimes excellent vehicles of presentation to non-technical management [11]. Furthermore, most models have a rich theoretical background that makes numerous tasks such as generation of large test suites easy to automate [12].

SUT fault detection

MBT is more likely to expose failures in the SUT because of the variety of tests that can be generated using models. Examples are failures caused by exhaustive combinations of inputs, failures that are revealed only over time (memory leaks), and failures that are revealed by exercising different combinations of variables (whatever is described in the model) [11]. Comparative studies [4][10][13][28] show that MBT is as good as or better at fault detection than manually designed tests. However, its fault detection power depends on the skill and experience of those writing the model and choosing the test selection criteria [35].

Reduced testing cost and time

If the time needed to write and maintain the model plus the time spent on directing the test generation is less than the cost of manually designing and maintaining a test suite, MBT practices will lead to less time and effort spent on testing. It might also save time during the failure analysis stage after test execution. Firstly, because failures are reported in a consistent way and secondly, because some model-based tools are capable of finding the shortest possible test sequence that causes the failure. Thirdly, since not only the code can be inspected, but also the abstracted test cases which give an overview over the test sequence through the model [35].

Improved test quality

In manual testing the quality of tests is highly dependent on the engineer and the test design may not be reproducible. MBT however uses an automated test generator based on algorithms and heuristics to choose the test cases from the model, which makes the design process systematic and repeatable. Since the input data and the test oracles are generated from the model, the cost of generating more executable test scripts is just the computing time required to generate them [35].

Coverage

Coverage is used in various forms to evaluate either test progress or the adequacy of the generated tests. Coverage can also be expressed for a model. Model coverage is therefore another heuristic that provides insight into the thoroughness and effectiveness of the testing effort, especially when testing does not reveal failures [11]. Coverage typically deals with the control-flow through the model [35].

Requirements defect detection

Writing a model for testing may expose issues and defects in the informal requirements (see defect detection in Appendix B.1.2). The first step in MBT is to create an abstract model of the SUT, hence this phase typically exposes requirement issues. This is a major benefit of MBT because requirement problems are a major source of system problems [35].

Traceability

Traceability is the ability to relate each test case to the model, to the test selection criteria, and even to the informal system requirements. Traceability helps to explain the test case as well as why it was generated. Furthermore it can be used to optimize test execution as the model evolves, since it enables the possibility to execute just the subset of the tests that are affected by the model modifications. From an abstract view traceability is a relation between the elements of the model and the test cases [35].

Requirements evolution

In manual testing significant efforts are often required to update the test suite as the requirements of the system changes. When using MBT only the model has to be updated and the tests regenerated. Since the model is usually much smaller than the test suite, time is saved when updating the model compared to updating all tests manually, resulting in faster response to evolving requirements [35].

2.5.6 Limitations

No system comes without drawbacks or limitations. MBT is no different. A fundamental limitation of MBT is that it cannot guarantee to find all the differences between the model and the implementation, even if generate a very large test set. However, this is a limitation for all kinds of testing [35]. Limitations of MBT are thus discussed below.

Tester skills

A practical limitation of MBT, at least initially, is that different skills are required compared to manual test design. The model designers must be able to abstract and design models, in addition to being experts in the application area. This requires training costs and an initial learning curve when starting to use MBT [35].

Sizeable initial effort

With the exception of the initial effort required when deploying MBT, in terms of the required tester skills and other allocations for making preparations, there is a sizeable initial effort for each testing process in order to save resources at various stages later in the testing process. Selecting the type of model, abstracting system functionality into multiple parts of a model, and finally building the model are all labor-intensive tasks (9).

Testing characteristics

MBT is usually used for functional testing, which is a limitation. There is little experience using MBT for other types of testing, such as performance testing and robustness testing. Some types of testing are not easily automated, such as testing the installation process of a software package. These are better tested manually [35].

State space explosion

There are drawbacks of models that cannot completely be avoided. For state models (and most similar models) the most prominent problem is state space explosion. Models of any non-trivial software functionality can grow beyond manageable levels. Almost all other model-based tasks, such as model maintenance, checking and reviewing, non-random test generation and achieving coverage criteria, are affected in this scenario [12].

Time to analyze failed tests

As a generated test fails, it must be decided whether the failure is caused by the SUT, the adaptor code, or an error in the model. This is similar to manual testing, where it has to be decided whether the failure was due to a fault in the SUT or in the test script. MBT however generates test sequences that might be more complex and less intuitive than manually designed test sequences. Thus, it might be more difficult and time-consuming to find the cause of the failed test [35].

Traditional metrics

In the manual test design process often a number of measures are used to measure the testing progress, for example the number of test cases designed. Such measures are not useful when applying MBT, since the approach can generate huge numbers of test cases. Measurements of test progress should instead move towards other measurements, such as SUT code coverage, requirements coverage and model coverage metrics [35].

Outdated requirements

Another consideration when adopting MBT is that requirements tend to be outdated. As software project evolves the informal requirements sometimes become out of date. If this would apply when using MBT, the wrong model will be built and test case execution will yield a significant amount of errors in the SUT [35].

Inappropriate use of model-based testing

Some parts of the SUT may be tested more effectively and quicker by designing a few test cases manually. The risk is that it takes some experience of MBT usage to know which aspects of the SUT should be modeled and which should be tested manually, but also to know which types of applications that conforms well to the model-based approach [35].

2.5.7 Summary

This section defined and explained the concept of MBT, starting with a background and an introduction.

MBT is really a collection of black-box testing techniques, thus different approaches were described. Four different approaches were described and they primarily differ in which phases of the testing process they automate.

This section continued by describing the general MBT process, which includes creating a model of the SUT, generating abstract test cases, concretizing those abstract test cases, execute test cases and finally analyzing the test results. This may be compared against the classical testing processes described in section 2.4.2, in terms of which parts of the testing process are automated.

Table 2.1 below summarizes the different approaches to test automation in terms of different testing processes (see section 2.4.2 for discussion of classical testing processes).

Testing Process	Test phases			
	Test cases	Test execution	Test coverage	Test result analysis
Manual	<i>Manual design</i>	<i>Manual execution</i>	<i>Manual analysis</i>	<i>Manual analysis</i>
Capture/Replay	<i>Manual design</i>	<i>Automated execution (records manual execution)</i>	<i>Manual analysis</i>	<i>Automated analysis (manually written)</i>
Script-based	<i>Manual design</i>	<i>Automated execution</i>	<i>Manual analysis</i>	<i>Automated analysis (manually written)</i>
Keyword-based	<i>Manual design</i>	<i>Automated execution</i>	<i>Manual analysis</i>	<i>Automated analysis (manually written)</i>
Model-based	<i>Automated design (generated from model)</i>	<i>Automated execution</i>	<i>Automated analysis (generated from model)</i>	<i>Automated analysis (generated from mode)</i>

Table 2.1: Overview of testing processes and approaches to test automation

As Table 2.1 describes a manual testing process is completely manual.

The capture/replay approach only partially automates the test execution since it captures manual operations. This approach is still dependent on manual test execution, but records the session which later can be re-run. Small changes in system functionality require the process of capturing manual operations to be repeated.

The script-based and the keyword-based approaches are similar. Test case design and analysis of test coverage in terms of functional requirements are still manually performed. The test execution is automated in both approaches by using scripts which initializes the SUT, sets inputs, executes the SUT and captures the output. The test result analysis phase may also be automated. However the expected behavior and outputs must be manually specified for the initial set of tests. The difference between the two approaches is that a keyword-based testing process raises the abstraction level by using keywords for test case design, thus expressing each test case as abstractly as possible. The goal of the keyword-based approach is to overcome the maintenance issue that comes with a script-based testing process.

The model-based approach solves most of the issues of the other approaches. Test case design is automated by generating test cases from a behavioral model. The generation of test cases also solves the test execution problem since generated abstract test cases are rendered to executable test scripts. Also, both the test coverage analysis and the test result analysis are automated because this information is generated from the model.

Sections 2.5.5 and 2.5.6 discussed benefits and limitations of MBT.

2.6 Interfaces

This thesis will involve several interfaces. These are described briefly in this section. Qtronic (the MBT tool) and the test object used for evaluation will be described in more detail in Chapter 3.

2.6.1 Test object (ATM)

The test object that will be used in the thesis is a simplified model of an ATM system, modeled as a client-server application.

Wikipedia defines an ATM as [37]:

“An automated teller machine (ATM) is a computerized telecommunications device that provides the clients of a financial institution with access to financial transactions in a public space without the need for human clerk or bank teller.”

The test object will be discussed in greater detail in Chapter 3. See Appendix C.1 for specifications.

2.6.2 Qtronic

This thesis focuses on evaluating MBT as a concept as well as a tool designed for this purpose. The tool used is Qtronic, developed by Conformiq [7].

Qtronic is a tool for automatically designing and creating test cases. The tool uses high-level system models as its input and calculates a set of test cases mathematically. Test cases are then exported in user-definable formats, such as TCL, TTCN-3, Visual Basic, HTML, XML, or Python. The tool can be used as an Eclipse plugin or as a stand-alone application, and can run on various platforms such as Windows, Linux and other UNIX variants [7].

The main features of the tool are automatic generation of test plans and executable test scripts. It also provides traceability matrices, test case dependency information, human-readable test plans, graphical representation of tests as sequence charts, and a graphical mapping between the input models and the generated test cases. Finally, it also supports several black-box design techniques, such as boundary value analysis, atomic condition coverage, and orthogonal array testing. These are just a few of the features offered [7].

Qtronic will be described in greater detail in section 3.2.1 and in Appendix C.2.

2.6.3 Java

The test object, or the ATM system, is implemented in Java.

Java [24] is an object-oriented programming language which makes use of extra libraries of software for developing programs. In the context of this thesis Java is used to implement the system under test, or test object, used to evaluate the Qtronic tool and MBT as a concept.

2.6.4 TCL

TCL [36] is a string-based command language, or scripting language, that only has a few fundamental constructs. TCL is designed to act as the glue that binds software building blocks into applications. TCL is interpreted when the application runs.

In this thesis TCL is used for test scripts as well as for executing them. Qtronic renders the generated abstract test cases to executable test scripts in TCL. The procedures used in the test scripts then have to be defined and implemented in the test harness (or adaptor code), using TCL. Also, TCL is used for executing these test scripts and for retrieving output from the SUT (via a socket channel) to a log file.

2.6.5 UML

UML [29], or the Unified Modeling Language, is a modeling language for expressing object-oriented design models. It is really a unification of a number of earlier object-oriented modeling languages. UML describes a system from a set of views, which represent properties of the system from various perspectives and relative to various purposes. Views are presented in models, which define a set of model elements, their properties and the relationships between them. The model contains information which is communicated in graphical form, using various types of diagrams.

UML is used in this thesis to create state charts, or state machines, in the Qtronic Modeler [8] (a separate tool of Qtronic used to create graphical models which then are imported into Qtronic). The state machine logic of the SUT will hence be modeled graphically using UML.

2.7 Summary

In this chapter the concept of MBT was introduced. Since MBT is a new testing methodology a general description of software testing was presented as a background.

The chapter started by discussing requirements and specifications. Requirements and specifications are necessary for testing since they specify describe how a system should work. Thus they are crucial for software testing since specified and actual system behavior can be compared.

The next section of this chapter discussed software testing in more detail. There are different kinds of testing as well as many different testing techniques. This section was divided into testing levels, testing characteristics and testing techniques.

Testing levels are really levels of abstraction in terms of what is to be tested, and in what detail. The testing levels described include unit testing, component testing, integration testing and system testing, ranging from the smallest to the largest scope. Testing characteristics describe different types of characteristics subjected to testing. These include functional testing, performance testing, robustness testing and usability testing. Testing techniques describe different types of high-level test design techniques and describes the relationship between them. Test design techniques can be divided into static and dynamic testing at the highest level. Furthermore, dynamic testing can be divided into black-box and white-box techniques, where black-box techniques can be further divided into functional and non-functional testing.

Before MBT was introduced, a prelude was given in section 2.4. This section included a description of the general testing process as well as classical testing processes used in industry. This section also included a description of the testing process used at Tieto.

Section 2.5 described MBT. The description of MBT started with a background and an introduction to the concept. This section also described the scope of MBT. Since MBT is really a collection of black-box techniques, different approaches of MBT are described. Furthermore the MBT process was described as a sequence of phases. Finally, benefits and limitations of MBT were discussed in this section.

The last section of this chapter described different interfaces that will be used in this thesis. These interfaces include the MBT tool to be used (Qtronic), the test object to be used, Java (used to implement the test object), TCL (used for test scripts) as well as UML (used for modeling the SUT).

3 Experiment

This chapter describes the experiments performed in this thesis. The chapter starts by describing the purpose of the thesis experiments as a background to the work. The test system, including Qtronic (the model-based testing tool used), the test object, the test scripts and the test execution environment, is then described as an introduction to the different artifacts and interfaces used in the work.

In section 3.3 an introductory example using model-based testing (MBT) is given to define and describe the work process and the different tasks when applying MBT with Qtronic. This example includes further details compared to the description of the test system. Design factors related to model quality are discussed in section 3.4 as an introduction to the feasibility study. Section 3.5 includes the three experiments performed in this thesis.

3.1 Purpose

The general purpose of the thesis experiments was to evaluate MBT as a concept as well as a specific MBT tool, namely Qtronic. Specifically, the purpose was to develop a test object and to test this test object using Qtronic. The idea was that the test object would be extended and modified for different experiments while documenting results and experiences. Thus results and experiences could also be compared between for the different experiments.

3.2 Test System

This section describes and defines the test system, which includes the artifacts and the interfaces used in the thesis work.

3.2.1 Test tool: Qtronic

The MBT tool used in this thesis is Conformiq Qtronic 2 [8]. Qtronic automatically designs test cases given a model of the system under test as input. The generated test cases are black-box tests, which mean that they evaluate the system under test based only on the system's external behavior [8].

Before starting experimenting using Qtronic, a course on the tool was attended. The course was held by Conformiq in the Tieto test lab at Sätterstrand, Hammarö. The course included two instructors and lasted for three days. Furthermore, one of the instructors stayed for two additional days to assist in the initial phase of the thesis work. The instructor helped modeling the test object and provided valuable insight into the functionality of the tool (Qtronic) as well as for the modeling language (QML).

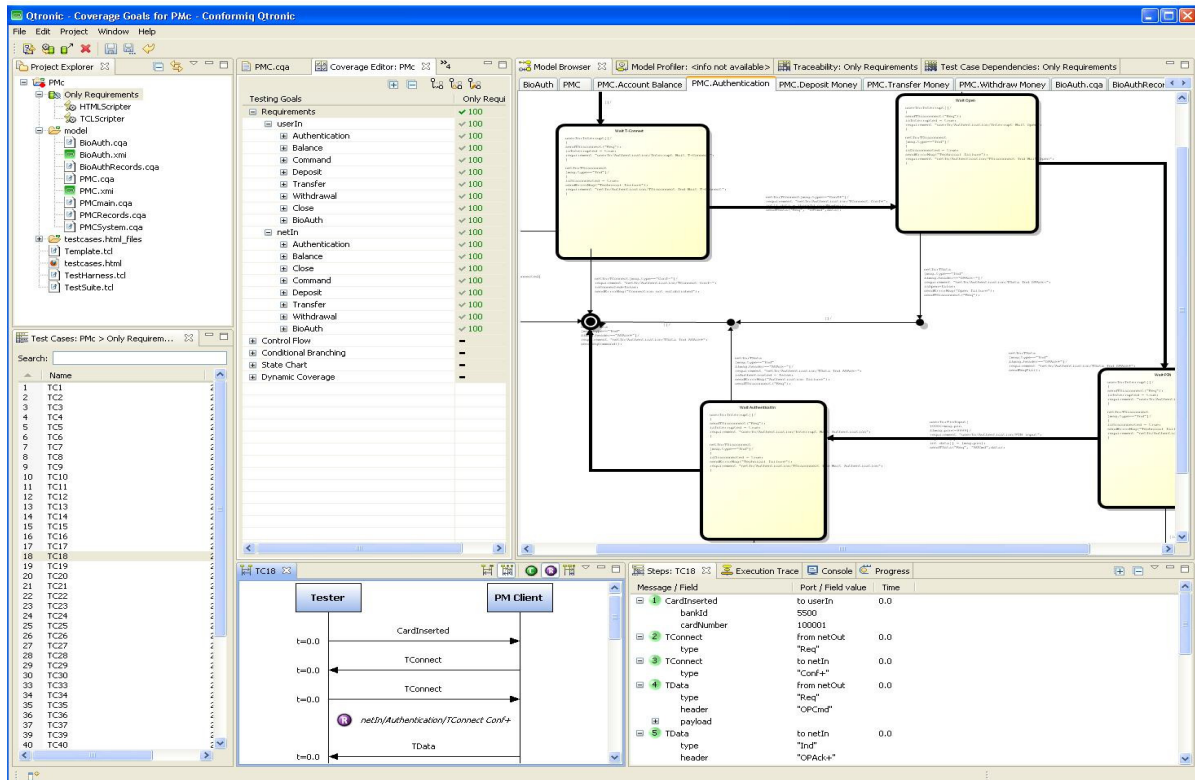


Figure 3.1: Qtronic stand-alone client

Figure 3.1 shows a print-screen of the Qtronic tool, specifically the stand-alone version. Qtronic can also be run as an Eclipse-plugin.

Model as input

The input to Qtronic is a model of the SUT. The model is a description of the intended behavior of the system. This model can be expressed as a collection of [8]:

1. Textual source files, which describe data types, constants, classes and their methods.
2. Graphical notation in form of state-chart diagrams with methods and procedures representing behavioral logic.
3. Class diagrams as a graphical alternative to declaring classes and describing class relationships using textual notation.

These models may be seen as behavioral or functional requirements. They describe the external characteristics of the system, thus the models do not need to reflect the actual implementation structurally given that they describe the intended outwardly observable behavior [8].

Qtronic supports multiple types of models, created and modified using different tools. Such tools include Qtronic Modeler (which is a separate tool shipped with Qtronic) as well as third-party modeling tools, such as Enterprise Architect and Rhapsody System Designer [8].

Qtronic Modeling Language (QML)

One way to express design models is the Qtronic Modeling Language (QML). Design models in QML can be expressed entirely in textual notation or together with graphical notation.

QML is an object-oriented language, which is based on Java, although it includes some ideas from C#. Compared to standard Java, the QML language is restricted or enhanced in a few ways. Examples of enhancements are support for global variables and global methods, structured value type records and operator overloading. Examples of restrictions are no support for enumerators, no support for packages and no support for annotations. In addition, the standard library of QML is very limited compared to the standard library of Java [8].

Qtronic Modeler

In addition to the textual notation, QML also includes graphical notation which can be used to create design models. A separate tool, Qtronic Modeler, is used to create these models in the QML graphical notation. The Qtronic Modeler is a simple tool for drawing UML state machine diagrams.

Creating state machines in the graphical notation defines the state machine execution logic, which otherwise has to be defined in the textual notation. The graphical state machines consist of states and transitions, as well as initial and final states. A state may also include an internal state machine. The transitions of a state machine are described using transition strings, consisting of three parts:

- Trigger
- Guard
- Action

A *trigger* is used to model the reception of an event and a *guard* is a Boolean condition for the transition to be executed. An *action* is executed on receipt of an event where a guard yields true. However, these three parts are not obligatory and an empty transition string is also valid. These three parts are all defined in the QML textual notation.

The graphical notation is always used with the textual notation. The very minimum textual notation is a class corresponding to the state machine defined in graphical notation and a system block describing the interfaces and the possible messages of each interface is required. Such messages have to be defined as value record types.

Qtronic Projects

Qtronic projects contain three types of information: model files, test design configurations, and test generation options. The model files can either be textual files defined in QML or graphical state diagrams, as described above.

The test design configurations may be used by the user to create different profiles with different coverage settings and different scripter plugins for different purposes. For example, the user may in one case want to create a test suite for basic requirements of the system, and in another case create a test suite for testing more detailed aspects, such as parameters by using boundary value analysis. The user can in this case create two distinct test design configurations for the project. The test design configurations contain a set of coverage settings. See Appendix C.2.1 for a description of these different coverage settings.

Test design configurations may include one or more scripter plugins for rendering the generated abstract test cases to an executable format. Scripter plugins for rendering test scripts in TCL, TTCN-3 and Perl are included in Qtronic. Test cases may also be rendered in HTML as test case documentation using the HTML scripter plugin.

The third project information in Qtronic is test generation options. These options are global across different test design configurations and apply to the project in general. Such test generation options include lookahead depth, maximum delay, only finalized runs, require conversion for interoperability testing, OSI methodology support and test case name prefix. These options are defined in Appendix C.2.2.

Test Case Generation

Qtronic uses a client-server architecture, where the client user interface is an Eclipse-plugin or a stand-alone version. The server component, Qtronic Computation Server, may run locally or remotely. The Qtronic Computation Server is used for test generation, which is a computationally intensive task. Therefore it is recommended running the server remotely [8]. However, in this thesis the computation server is run locally.

The first step in the test generation process is to load the model files to the computation server. Once the coverage settings for the test design configuration have been defined, the test generation may be started. When the test generation is started by the user in the client user interface, the test generation is triggered on the computation server. As the test generation progresses the console window in Qtronic, or in the Eclipse console window, will display the status of the test generation. Figure 3.2 below illustrates the test generation progress in Qtronic.

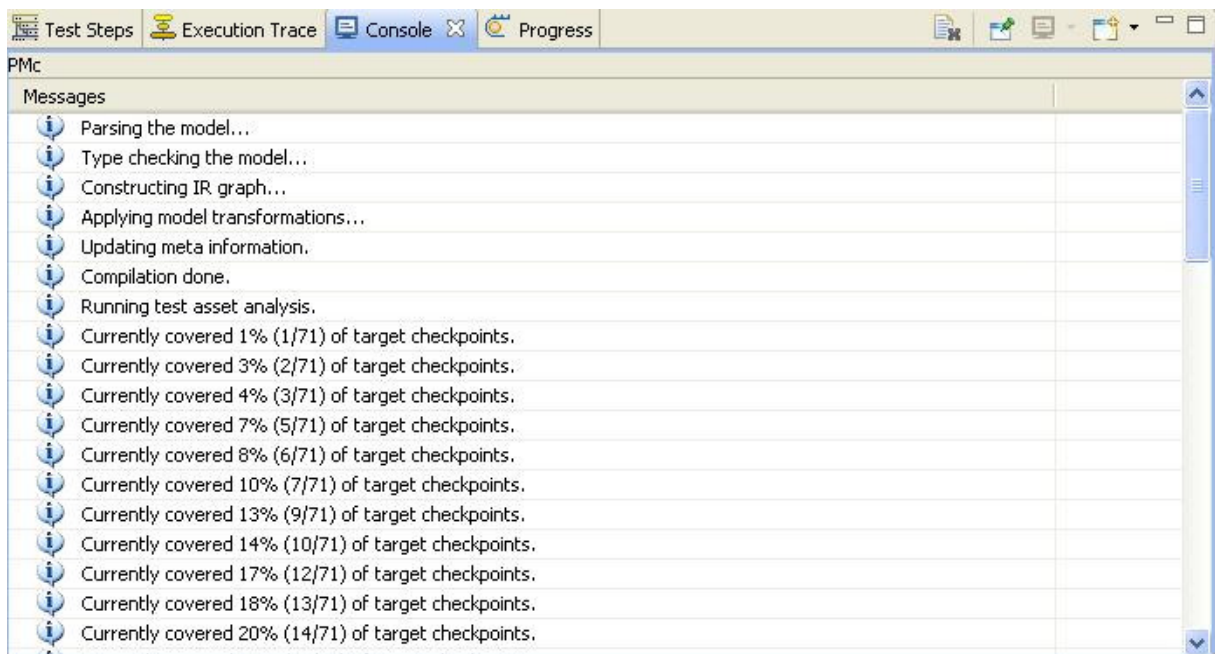


Figure 3.2: Status of the generation shown in Qtronic console

When the test generation is completed, and if successful, a set of test cases will be displayed in the tool. The test cases only exist within the tool and the project, and must be exported using a scripting back-end to be executable. The test case generations can be

inspected in a number of ways in Qtronic. See Appendix C.2.3 for brief descriptions of the available views.

Figure 3.3 shows an example of an interaction description for a test case.

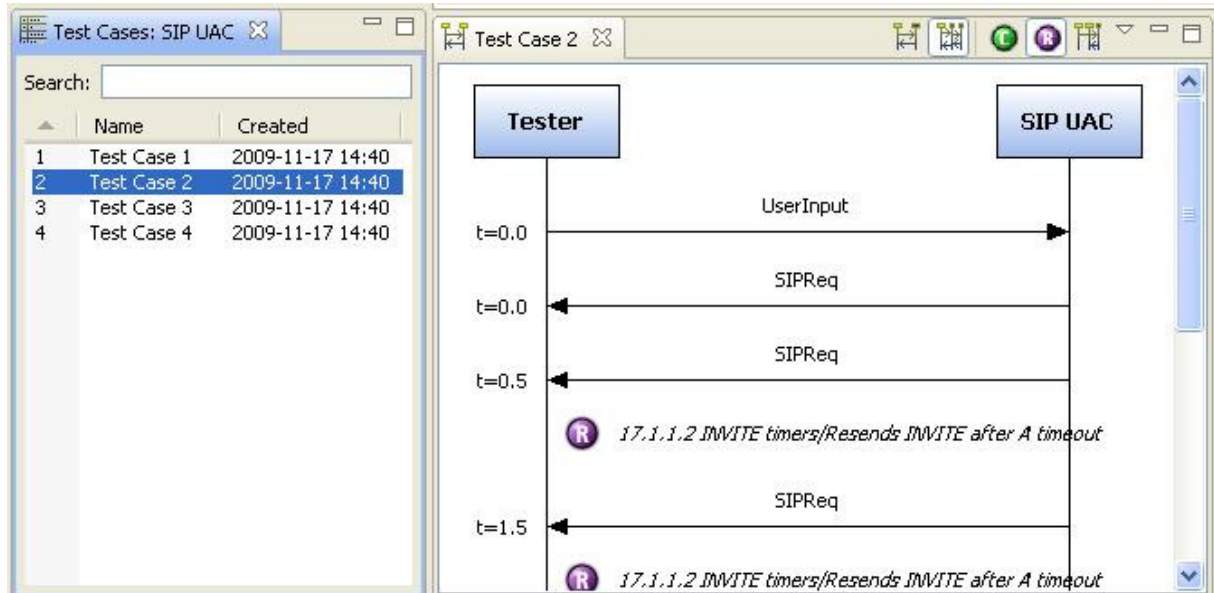


Figure 3.3: Test case interaction

The model used for test generation in Figure 3.3 describes partial functionality of a SIP User Agent Client, and includes call setup, call termination by caller or callee and call cancellation during call setup

Each Qtronic project may include several test design configurations. Hence, the generated test cases are grouped by the test design configuration used for test generation. For example, suppose that one test design configuration aims to cover all the states in the model, and the another test design configuration aims to cover all the transitions of the same model. When generating tests using these configurations two sets of test cases will be generated, one for each test design configuration. These two sets are hence independent of each other.

Script Generation

When abstract test cases have been generated in the tool, they may be rendered into an executable format using one or more scripting back-ends, or scripter plugins. Each test design configuration may contain more than one scripting back-end. Qtronic is shipped with scripting back-ends supporting test script generation for TCL, TTCN-3 and Perl. A scripting back-end for rendering test case documentation in HTML is also provided. Moreover, scripting back-ends may be defined by the user, using a plugin API (Application Programming Interface) defined in the Qtronic User Manual [8].

3.2.2 Test object

The test object model used in this thesis is a part of a simplified model for an ATM client-server system, using an application-layer protocol. The system includes the ATM (the client) and the central bank computer (the server), which communicates over a network.

Overview

Both the client and the server are divided into subsystems. The client consists of the user interface, where the user can withdraw money and request account balance information, and the protocol module. The server consists of the account database application and the protocol module. The client and server communicate over a fifth subsystem, the transport service provider (TSP).

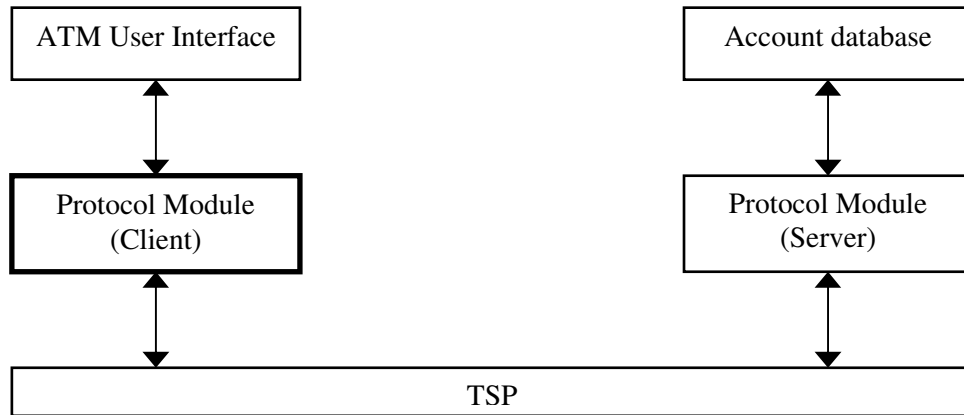


Figure 3.4: ATM system overview

As Figure 3.4 indicates the test object is limited to the protocol module of the client side. The client protocol module has two interfaces: one towards the user interface and one towards the network.

Protocol Module (Client)

The test object in this thesis was chosen to be the protocol module of the client since data communication protocol includes the most complexity. The scope is limited to the protocol module of the client. The reason is that also modeling and implementing the communication protocol of the server would be similar to the communication protocol of the client and not really make any further contribution. Also, if both protocol modules were to be implemented only really the user application interface of the client and the database application interface of the server would be tested. The generated test cases are black-box tests, thus the black box would be the client and the server. The generated test cases would only include input and output for those application interfaces and not for the network interfaces. Moreover, if the two protocol modules were implemented but not communicating they would only really be two separate components, of similar complexity, being tested.

The protocol module is a state machine that handles incoming messages (events) and responds with outgoing messages (actions) for both interfaces. Messages either come from the user interface or from the network, and the corresponding actions are sent to the user or to the network.

See Appendix C.1.1 for the state transition diagram and further details of the client protocol module.

Implementation

The protocol module is implemented in Java, according to the state design pattern [15]. The state design pattern uses a class *Context* which defines the user interface and a base class *State* which defines the interface for encapsulating the behavior associated with a particular state of the *Context*. Subclasses of *State* implement a particular behavior associated with a state of the *Context*. Also, the *Context* maintains an instance of a subclass that defines the current state [15].

The implementation of the test object follows this pattern and makes use of inheritance and polymorphism. The *Context* class maintains one instance of *State* and uses polymorphism to access the instances of the subclasses (i.e. the current state). Instances of the subclasses are also maintained in the *Context*. Each state described in the state transition diagram (see Appendix C.1.1) is implemented as a subclass.

3.2.3 Test scripts

In this thesis the abstract test cases generated in Qtronic are rendered to executable test scripts in TCL (see section 2.6.4). Qtronic generates the following TCL files when rendering test scripts in TCL:

- Test case template (Template.TCL)
- Test harness (TestHarness.TCL)
- Test suite (TestSuite.TCL)

The test case template file is empty when generated. According to the Qtronic user manual [8] extra code, such as initialization and de-initialization of the test harness, may be inserted. However, in this thesis this file has not been used.

The test harness file is the library file which contains the implementation of the procedures that the scripter generates. The generated file contains empty procedures, which must be implemented. Each procedure corresponds to input or expected output from the SUT, as modeled. Thus, procedures corresponding to modeled input send data to the SUT. Procedures corresponding to modeled output receive SUT output and compare the actual output to the expected output. Hence, procedures corresponding to SUT output implement verdict functionality, i.e. compare expected and actual output.

The test suite file contains all the test cases. Each test case is defined as a procedure and the generated code uses the procedures generated in the test harness file to define each test case. Each test case procedure is a sequence of parameter initializations and procedure calls (to procedures defined in the test harness). The test suite file is completely generated by Qtronic, meaning that it requires no implementation or modification.

The procedures in the test harness may be considered as keywords used to define the test cases (compare to the keyword-based testing process in section 2.4.2). Each procedure has zero or more parameters, depending on the design model used for generating the test cases. The number of generated procedures depends on the system block, which is an obligatory part of the model in QML. For example, if four different types of messages have been defined as incoming messages of the network; four procedures are generated, where the procedure name is a concatenation of the interface name and the message name, as specified in the system block.

These test harness procedures are implemented after the test generation. The implementation of each procedure includes handling potential parameters as well as sending input to the SUT or retrieving output from the SUT. For the retrieved SUT output comparisons can be made to the expected output. This expected output is defined in the test case and consists of the test harness procedure name (each procedure corresponds to a specific message type) and the procedure parameters. All procedures generated from an outbound interface, either outgoing messages to the user interface or to the network, will be retrieving output from the SUT and comparing against expected values. Correspondingly, all procedures generated from an inbound interface, i.e. incoming messages, will be sending input to the SUT.

The implementation of the test harness procedures depends on the test execution environment, i.e. how to communicate with the SUT. When the procedures in the test harness have been implemented the test cases can be executed independently. The test harness defines only the individual procedures used for defining the test cases, and not the test execution environment. However, the implementation of these procedures makes use of the test execution environment.

3.2.4 Test execution environment

The test execution environment consists of socket communication between the test scripts and the SUT, running on the same machine. The test scripts use socket procedures implemented in TCL and the test object uses socket procedures defined in Java. The socket defined in Java is a server socket, whereas the socket used by the test scripts is a client socket.

Test Scripts

The procedures implemented in the test harness make use of socket procedures, defined in a separate TCL file. This file includes procedures for initializing the socket channel, sending data and receiving data over the socket, as well as closing the socket. As described in section 3.2.3, the sending procedure and receiving procedure are used to implement the test harness. All procedures generated from the inbound interfaces are defined using the sending socket procedure. Correspondingly all procedures generated from the outbound interfaces are defined using the receiving socket procedure.

The test execution environment also executes the test cases, which are defined as TCL procedures, using a test execution script. This script also assigns a pass/fail verdict to each test case. The verdict of each test case is determined by the comparisons between expected and actual output (see a more detailed discussion below). Furthermore, this test execution script sets up the socket communication by initializing a client socket. The test execution environment as implemented in the test scripts also include log functionality, in terms of a TCL procedure that writes relevant information to a text file. The resulting log file includes input sent to the SUT as well as actual and expected output from the SUT. The log file also includes a verdict for each test case and hence also indicates mismatches between expected and actual SUT output. Thus this log file may be used to analyze the test execution.

To summarize, the test execution environment implemented in TCL consists of procedures for socket communication with the SUT, including procedures for sending and receiving data, initialization the socket channel and closing the socket. Furthermore the test execution environment includes TCL procedures for log and verdict functionality.

Verdict functionality

The basis for assigning pass/fail verdicts to the test cases is the expected and the actual SUT output. The generated test cases are black-box tests and are defined by sequences of SUT input and expected output. A certain SUT output is expected given a particular SUT input, according to the model. For example, each test case of the test suite file starts with a procedure call to a test harness procedure corresponding to a message type for SUT input, as modeled. The next procedure call of the test case is always to a test harness procedure corresponding to an expected SUT output. Thus the success of an executed test case is dependent on the comparison between expected and actual SUT output.

The verdict functionality of the test execution environment implemented in TCL consists of a procedure that compares the expected and actual SUT output, which are strings. Thus the comparison is a string comparison. The actual output is retrieved on the socket channel whereas the expected output is set in each test harness procedure corresponding to modeled SUT output. The names of the generated test harness procedures are based on the interface and the message type, as modeled, on the form *<interface><message type>*. Hence the expected output is defined by the message type and the procedure parameters and set to a string, on the same form as the output strings of the SUT. The retrieved output string (from the socket channel) and the defined string for expected output are sent as parameters to the verdict procedure, which performs the string comparison. If the two strings do not match the test case fails. The string containing the expected output may be considered the oracle information (see section 2.5.3).

Log functionality

The log functionality of the test execution environment consists of a TCL procedure. The procedure is used in all test harness procedures and creates a log file during test execution. This log file contains all the input sent to the SUT as well as all expected and actual SUT output. Since the test suite contains one or more test cases the start and end of each test case is indicated in the log file, as well as the name of each test case. Furthermore, the log functionality indicates mismatches between actual and expected output strings by writing “Output mismatch” to the file. Finally, a pass/fail verdict is included in the end of each test case in the log file.

Test Object

The test object is executed using a driver class. This driver executes the test object and uses a separate class, *Socket*, for receiving input and sending output to the test scripts. The driver class uses a server socket, which waits for requests to come from the network. These requests come from the test scripts.

The received input is forwarded to the test object, which triggers events within the state machine. Every triggered event results in at least one action, which is returned to the driver. The driver then sends the resulting actions, really the output of the SUT, back to the test scripts. Hence expected and actual output can be compared in the test scripts.

3.3 Introductory example

As an introduction to Qtronic and the different tasks involved in the process of using Qtronic, an example is given. In this example a subset of the protocol module is used, namely the account balance functionality of the ATM. Since the model used in this example is a subset of the complete model the test cases will still be valid for one of the SUT versions used in this thesis.

The process described in this example applies to the thesis work in general. This example is given to aid the understanding of the thesis work and to describe the way of working.

3.3.1 Modeling

This example models the functionality of the client protocol module (as described in section 3.2.2) for requiring the account balance through an ATM. The example is a subset of the complete model used in experiment 2 (see section 3.5.2) of this thesis. The specification used for modeling this functionality is specified in Appendix C.1.2.

The first step when testing with Qtronic is to create the model. An obligatory part of the model is the system block. The system block defines the interfaces, or the ports, of the model. This includes specifying what types of messages than can be sent and received. Ports are classified as inbound or outbound. The system block used in this example is defined below.

```
system {
  Inbound userIn: CardInserted, PinInput, BalanceQuery, Interrupt;
  Outbound userOut: RequestPin, RequestCmd, BalanceInfo, ErrorMessage;
  Inbound netIn: TConnect, TData, TDisconnect;
  Outbound netOut: TConnect, TData, TDisconnect;
}
```

The above specified messages are defined as value type records (structs in C). Defining the types of messages is really a design issue. Messages may carry additional information or simply be a named value type record without any data members. An example of a value type record definition for an incoming message from the user application (the ATM machine) is given below.

```
record CardInserted {
  int bankId;
  int cardNumber;
}
```

The other records defined for userIn and userOut follow the same pattern, although some do not contain any data members. However, the records defined for the network interface are more generic. An example is given below.

```
record TData {
  String type;
  String header;
  int[] payload;
}
```

The other records for the network interface, TConnect and TDisconnect, only specify the type. In this model the type is always Ind, denoting an indication, for incoming messages and Req, denoting request, for outgoing messages. Correspondingly, the header for TData is always an acknowledgement for incoming messages and a command for outgoing messages.

The next step is to model the behavior of the system under test. To be able to do so, a specification is required. The behavioral model is specified in QML and may be described using only the textual notation. However, in this example the Qtronic Modeler and the graphical notation (really UML) are used to create this model. This graphical model also includes blocks of QML textual notation, describing the logic of the transitions. Furthermore, textual notation other than in the graphical model is required. This includes a class corresponding to the top-level state machine of the graphical model, a system block and value type record definitions. The system block definition and an example of a record definition of this model are given above. Sample code of the QML state machine class is given below.

```
class ATMBalance extends StateMachine {
    public ATMBalance(){}

    public void sendTData(String type, String header, int[] payload){
        TData r;
        r.type = type;
        r.header = header;
        r.payload = payload;
        netOut.send(r);
    }
    //...
}
```

This QML class corresponding to the top-level state machine of the graphical model includes class methods and data members. The methods and the data members are in this model used to describe the transition logic in the graphical model, where they have global visibility.

Figure 3.5 below defines the top-level state machine for the model of the protocol module. The model in this example is limited to functionality for requiring the account balance (see Appendix C.1.2 for the specification).

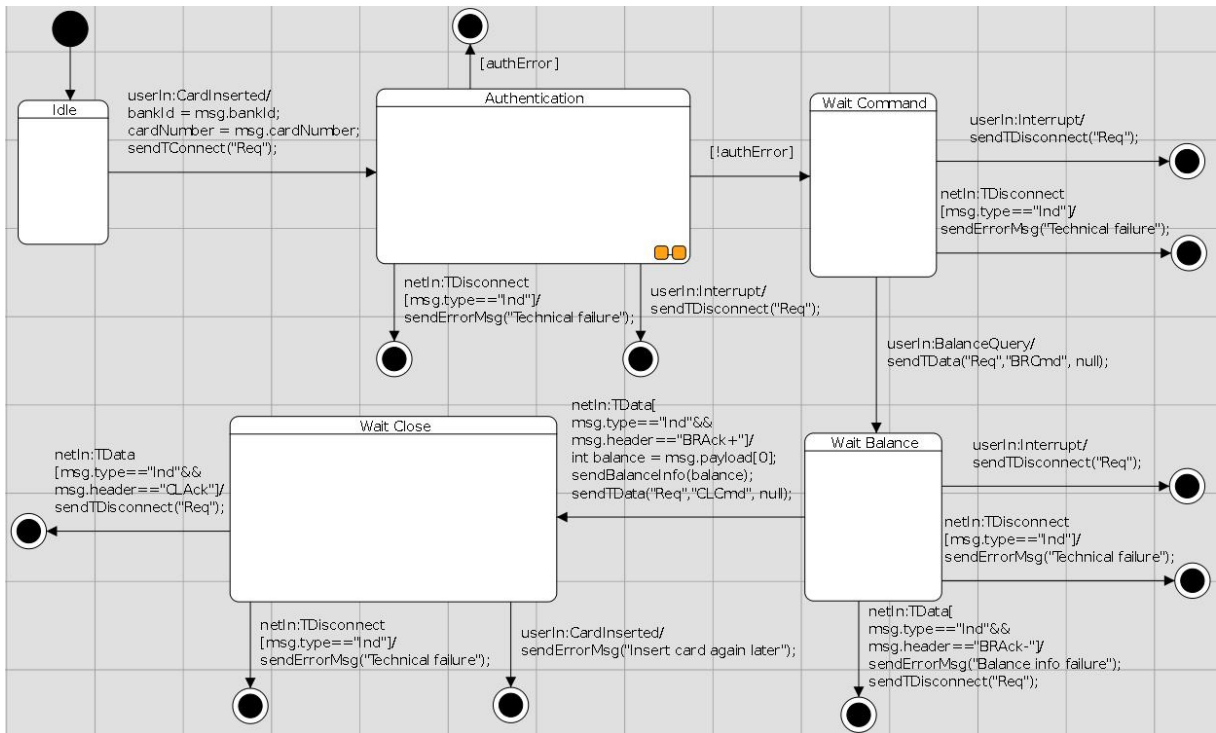


Figure 3.5: Qtronic example: Top-level state machine

The transition logic consists of three parts and is described as following: `<trigger>` [`<guard>`] / `<action>` (see section 3.2.1 and the Qtronic Modeler paragraph for a description of these). All three parts are optional, but a guard, if used, has to be placed within square brackets and actions, if used, has to be subsequent to a “/”. The trigger is defined by specifying the port and the message type on the form: `<port> : <message type>`.

The QML model in Figure 3.5 includes error handling which is not specified in the UML diagrams (see Appendix C.1.3). This error handling include user entered interrupts, unexpected disconnect indications from the network, negative acknowledgements and other unexpected events, such as inserting the card again before the ATM has output the account balance receipt. Acknowledgements from the server are only expected following a request from the client to the server. Such a request could be “BRCmd”, which stands for Balance Request Command, and is replied to with “BRACK+” if the request is successful. The model also includes an internal state machine for the state Authentication, which is defined in Figure 3.6 below (compare this with the specification in Appendix C.1.2).

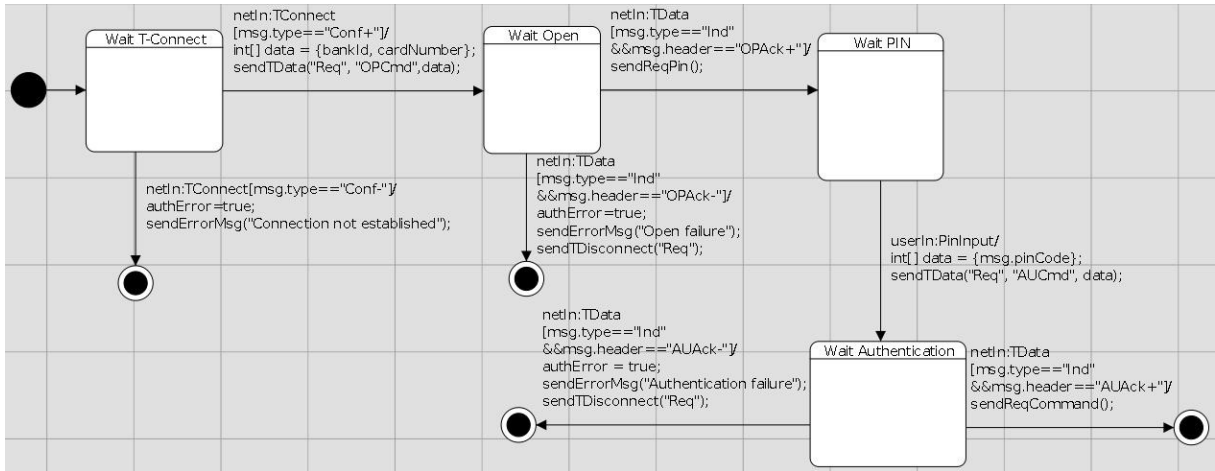


Figure 3.6: Qtronic example: Authentication state machine

The internal state machine of the Authentication state does not include error handling for user interruptions and network disconnections. Such events are handled at the higher level, for the Authentication state (see Figure 3.5), thus implicitly also for the internal state machine. This internal state machine is started when the Authentication state is entered in the top-level state machine.

3.3.2 Test generation

The first step in test generation is to load the model files to the computation server, which parses and type checks the model. This may either be done in a separate step or automatically by the tool before generating test cases.

Before generating test cases, a test design configuration needs to be specified (see section 3.2.1 and Qtronic Projects). This configuration specifies the coverage goals for the test generation. The different coverage settings and parameters are described in Appendix C.2.1. Figure 3.7 shows the test design configuration used in this example.

Testing Goals	*Test Design Configuration
<input checked="" type="checkbox"/> Control Flow	-
<input checked="" type="checkbox"/> Conditional Branching	-
<input checked="" type="checkbox"/> State Chart	- 0
<input checked="" type="checkbox"/> States	✓ 0
<input checked="" type="checkbox"/> Transitions	-
<input checked="" type="checkbox"/> 2-Transitions	-
<input checked="" type="checkbox"/> Implicit Consumption	-
<input checked="" type="checkbox"/> Dynamic Coverage	-

Figure 3.7: Qtronic example: Test Design Configuration

For simplicity, the test coverage is in this example set to cover all states in the model, including all final states. Furthermore, in the properties of the Qtronic project different algorithmic options are specified. These options are described in Appendix C.2.2. The algorithmic options are set to use a lookahead depth of 1 (the lowest possible value) and to only allow finalized runs. The lookahead depth corresponds to the number of external input events to the system or timeouts [8]. The only finalized runs option specifies that test cases

have to end in a final state of the top-level state machine [8] (see Appendix C.2.2 for further details).

During test generation the progress is displayed in the Qtronic Console. The test generation progress of this example is shown in Figure 3.8 below.

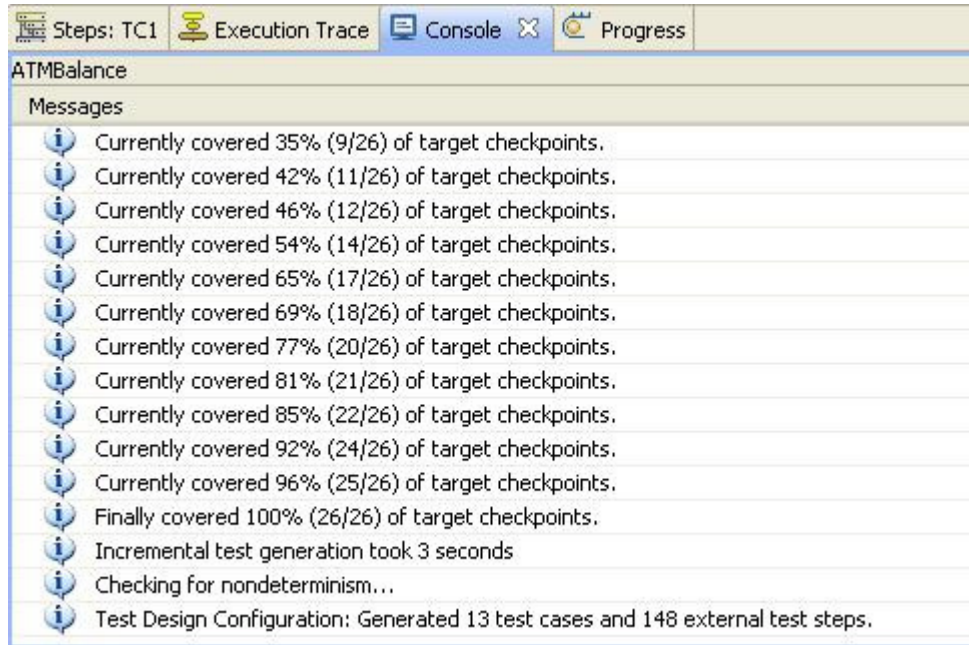


Figure 3.8: Qtronic example: Test generation progress

As illustrated in Figure 3.8, all states were covered in the test generation. Qtronic generated 13 test cases in 3 seconds. The test generation time depends on the complexity of the model, the test design configuration and the Qtronic algorithmic settings. An example of the results of the test generation, i.e. the test cases, is shown in Figure 3.9.

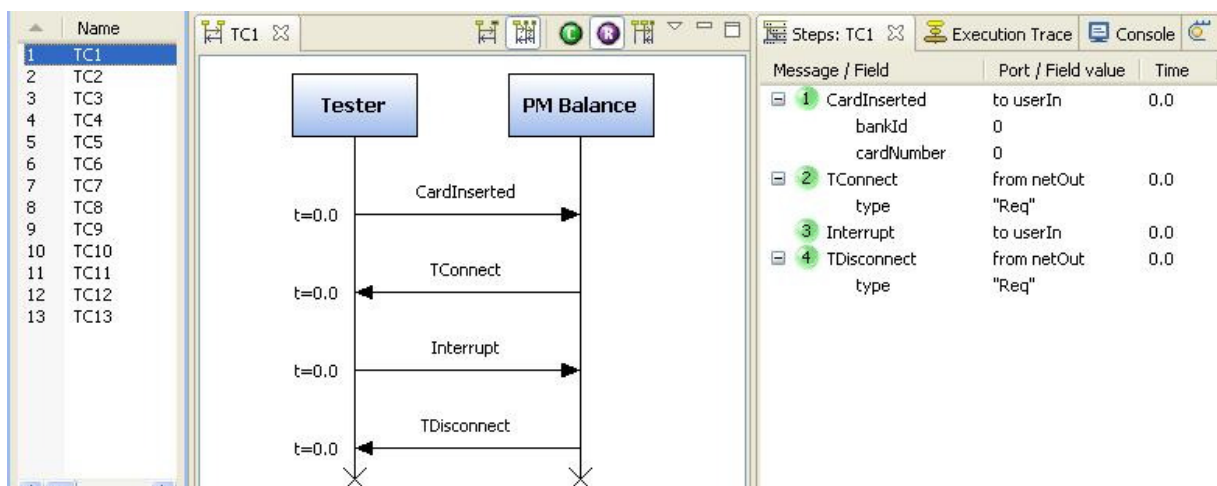


Figure 3.9: Qtronic example: Generated test cases

The left-most view in Figure 3.9 is a list of the generated test cases. The view in the middle describes the communication between the tester and the system under test as a form of

sequence diagram for a particular test case. The right-most view describes the same test case in further detail, including parameter values and which port the message was sent to or from. Qtronic also generates a traceability matrix, which in this example shows which states the different test cases cover (see Appendix B.3 for an example of a Qtronic traceability matrix).

3.3.3 Script rendering and test harness implementation

At this point test cases have been generated in Qtronic. However, the test cases are abstract and not executable. A scripting back-end is added to the test design configuration to render the test cases to test scripts. The scripting back-end is essentially a plugin written in Java that renders the abstract test cases to executable test scripts. In this example a TCL scripting back-end is used, which generates two files of interest: the test suite and the test harness (see section 3.2.3 for further details about these files).

```
proc "TC1" {} \
{
  set bankId_1 0
  set cardNumber_2 0
  userInCardInserted $bankId_1 $cardNumber_2
  //...
}
```

The code above is a sample from the generated test suite file. Each test case is defined as a procedure in TCL. The remaining definition of each test case follows the same pattern as illustrated in the code, as a sequence of procedure calls and necessary parameter initializations. The test case definitions uses the procedures generated in the test harness file. However, when generated, the procedures in the test harness file are empty and need to be defined manually. Examples of these procedure definitions are given below.

```
proc userInCardInserted { bankId cardNumber } { \
  global socketChannel
  set msg "Card inserted / $bankId / $cardNumber"
  send $socketChannel $msg
}

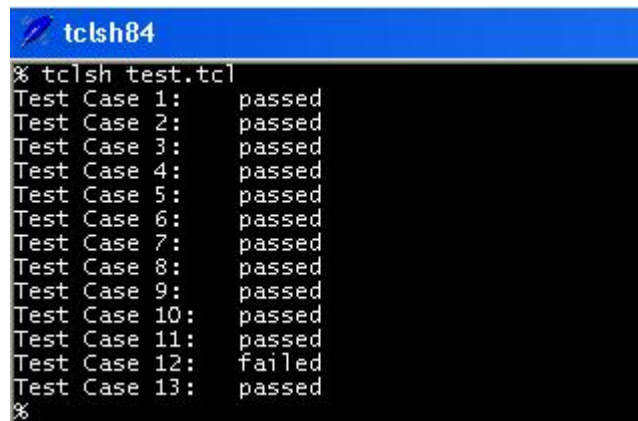
proc netOutTData { type header payload } { \
  global socketChannel received expected
  set expected "T-Data $type $header"
  if { $payload != "{}" } {
    set l [split $payload ,]
    for {set i 0} {$i < [llength $l]} {incr i} {
      set expected "$expected / [lindex $l $i]"
    }
  }
  vwait received
}
```

These two procedures are chosen for a reason, since the first procedure sends a message to the SUT, and the second receives output from the SUT. The test harness procedures either send input or receive output from the SUT. These procedures are really generated from the system block, as defined in section 3.3.1. For each defined port every specified message type will generate a procedure in the test harness file on the form `<port name><message type>`. The procedures generated for an inbound port will send input to the SUT. Correspondingly all

procedures generated from an outbound port will receive output from the SUT, and also be able to compare the output with the expected output generated from the model. This comparison is done in a separate procedure when receiving output on the socket channel, simply by comparing the two strings (the *received* string with the *expected* string defined in the code above). If a mismatch occurs this is written to a log file. A test case passes given that no mismatches occur.

3.3.4 Test execution

The next step is the test execution. The procedure for executing the test cases is to first start the SUT in Eclipse. The SUT will then wait for a socket request, which is sent by the test scripts (as described in section 3.2.4). The procedures defined in the test harness file uses other TCL procedures, namely procedures for socket communication. The test suite is executed in a separate script by a sequence of procedure calls for all the test case procedures. The result of this test execution is shown in Figure 3.10 below.



```
tclsh84
% tclsh test.tcl
Test Case 1: passed
Test Case 2: passed
Test Case 3: passed
Test Case 4: passed
Test Case 5: passed
Test Case 6: passed
Test Case 7: passed
Test Case 8: passed
Test Case 9: passed
Test Case 10: passed
Test Case 11: passed
Test Case 12: failed
Test Case 13: passed
%
```

Figure 3.10: Qtronic example: Test execution

The test case that failed was due to an error in a receiving test harness procedure, in form of an extra character. Hence the expected output differed from the actual output. A log file, containing a sequence of all SUT inputs as well as expected and actual output of the SUT, is generated during test execution. As mentioned, if a mismatch between the expected and actual SUT output this is indicated in the log file (see the log and verdict functionality described in section 3.2.4).

3.4 Design Factors in Modeling

As discussed in section 2.5.1 and as stated by Apfelbaum and Doyle [2], modeling is a good practice for capturing knowledge about a system and then reusing this knowledge as the system grows. Models may be used as a means of communication between different teams in an organization during development. For test teams, models provide a mechanism for structured analysis of the system. The greatest benefit of models is in reuse since the work done is not lost [2].

Models of complex systems may also be complex and hard to read since they may consist of a very large number of artifacts. Analysis of such models may be very difficult. Applying design guidelines when modeling makes this process easier. These design guidelines consist of rules, constraints and considerations for model construction. Such guidelines increase readability of models. Design conventions and guidelines have a positive impact on the model quality. Awareness of such modeling aspects has reported to result in increased readability, maintainability and understandability of models in the software development process. Well designed models have also been reported to aid and improve the communication between team members [16].

Design factors, or quality requirements of the model, are important considerations when modeling since models are reused, modified, maintained and extended. It is desirable that modifications and extensions to a model require as small changes as possible. That is, large modifications may be inevitable but the goal should be to create a model with a good structure that supports extensions and modifications. Since systems often are developed incrementally and may have a long lifetime, quality requirements such as extendibility and maintainability should be considered when creating models. Furthermore, as stated above, models may be used as a means of communication. Thus considerations and quality requirements such as readability are important. Since the model is the most important artifact of MBT, design factors should be considered and applied from the start when modeling a system. Readability will be considered since it is the quality requirement that is most closely related to the use of the model as a means of communication. Extendibility will be considered during the feasibility study since incremental development of complex systems often includes extending the system. Maintainability will be considered since maintenance often is a large factor in projects including complex systems.

Readability is defined as “the ease of understanding or comprehension due to the style of writing [22].” Extendibility is defined as “the ease of which a system or component may be modified to increase its storage or functional capacity [18]”. Maintainability is a quality factor which describes whether and how easily developers and users can upgrade the system, [19]. Furthermore, maintainability is defined as an attribute that relates the amount of effort required to make changes to artifacts [19].

3.5 Experiments

The methodology and procedure described in the introduction example (section 3.3) applies to the thesis work in general and gives an overview of the steps necessary and tasks involved in performing MBT with Qtronic. In the remaining parts of this chapter, the descriptions of the different experiments will not be discussed at the same level of detail.

In this thesis four experiments were performed. The approach of the thesis work was to incrementally develop the model and the test object for each experiment as if MBT were to be deployed in a new project within the organization. Since the experiments involved incremental development the first experiment was compared to the second experiment, the second to the third and finally the third was compared to the fourth.

The first experiment was to create a test object from the initial specification and then test the implementation using Qtronic. The second experiment used the resulting artifacts of the first experiment, both in terms of the Qtronic model, the test harness and the test object. New requirements were introduced and implemented without changing the existing functionality, and then tested. The third experiment was to change the general requirements for the authentication process, using the model, the test harness and test object resulting from the second experiment. The fourth experiment may be viewed as a second version of the third experiment since the specification was not modified for the fourth experiment. In that experiment logic were removed from the model of the third experiment and instead implemented in the test harness.

The descriptions of the experiments will primarily focus on the modeling because that is the most important task of the process. The test generation, the test harness implementation and the test execution follows the same pattern for all experiments and are not significantly changed between experiments, except for the fourth experiment.

3.5.1 Experiment 1

The initial experiment used the original specification (as defined in Appendix C.1.1). The experiment may be divided into sub-tasks. The first was to implement the test object according to the specification. The second task was to create a QML model describing the functionality of the client protocol module according to the specification. The remaining sub-tasks were to generate test cases from that model, render executable test scripts in TCL, implement the test harness procedures and finally execute the test scripts against the test object (the SUT). The tasks follow the way of working as described in the introductory example (see section 3.3).

Goal

The goal of this initial experiment was to apply MBT (Qtronic) to an existing test object and to execute the generated test scripts against that test object (the SUT).

Implementation

When starting this experiment the test object was implemented according to the initial specification (see Appendix C.1.1). The test object was implemented in Java and the implementation was based on the state design pattern [15] as described in section 3.2.2. The implementation also included error handling for negative acknowledgements, user entered interruptions and network failures.

Modeling

The modeling for this experiment was started immediately following the Qtronic course (mentioned in section 3.2.1). One of the course instructors stayed for two additional days following the course and assisted with the modeling to get the thesis work started. The instructor gave advice on which practices to apply when modeling in QML as well as about tool specific features. The QML model was completed during these two days and tested in the tool, i.e. loaded to the computation server and parsed. The model was also evaluated in a sense within the tool as a form of informal review (see Appendix B.2.1) against the specification.

The QML model included a top-level state machine called PMC (Protocol Module Client). This state machine made use of two internal state machines; Authentication and Withdrawal, as described in the specification (see Appendix C.1.1). The account balance functionality only included one state. Consequently no state machine was created for this functionality. Each state machine of the model had its own execution thread. The internal state machines are internal logic of a state of the higher-level state machine. The execution thread of an internal state machine is started when the state of the higher-level state machine is entered. The implication of the designed internal state machines is that they must be completed, i.e. reach a final state, before the top-level state machine can continue its execution. In this model a minimal number of final states were used, meaning that each state machine only included one final state (compare with Figure 3.5). Therefore all possible paths through the model ended in the same final state of the top-level state machine. Final states are not required in the model but they are necessary to ensure that the generated test cases are complete paths through the model, i.e. not ending in any state of the model.

The model made use of a number of QML procedures, defined within the class corresponding to the top-level state machine. These class methods were used to encapsulate action logic for the transitions, where the methods corresponded to the message types defined as outbound events in the system block for the model (see section 3.3.1). Thus some class methods, as in the example code below, were generic and often reused.

```
record TData {
  String type;
  String header;
}

class PMC extends StateMachine {
  public PMC(){}
  public void sendTData(String type, String header){
    TData r;
    r.type = type;
    r.header = header;
    netOut.send(r);
  }
  //..
}
```

The sample procedure and the message definition above illustrate a design choice for this experiment. Comparing this example with the *TData* structure and class method as defined in section 3.3.1, the payload field is not included in the record or the procedure. The focus of this experiment was to complete a working model using the UML diagrams in the specification and not the experiment with details such as parameter values. Hence most data members were omitted in the message definitions, such as card number and withdrawal amount. The only message records containing data members were the ones including a type or a header necessary to describe the message type as defined in the specification (see Appendix C.1.1). Consequently the test object used for this experiment did not handle data values but only the different message types. Figure 3.11 below include transitions of the model used in this experiment.

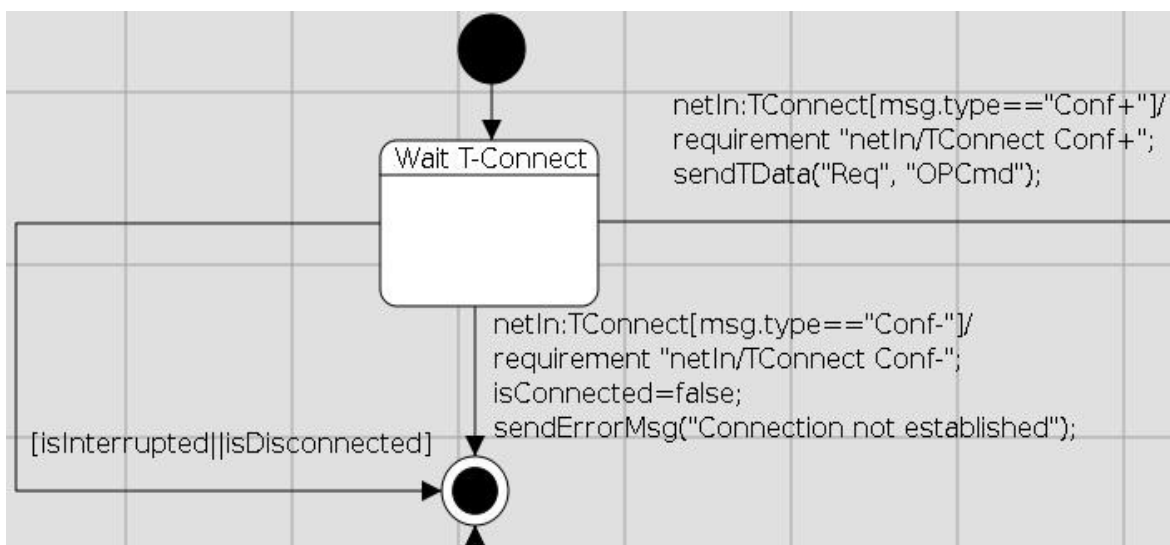


Figure 3.11: Experiment 1: Authentication state machine

One transition of the figure illustrates the use of the class method defined above and the type parameter of the message as discussed. The received message (the event) is bound to the local variable *msg* and is only visible inside the transition string (defined in section 3.2.1 under the Qtronic Modeler paragraph). The type of the transition event is in this case necessary for the flow of the model, since a connection attempt may be successful or unsuccessful and thus affects the path through the model. Such data members, or parameters, of messages were therefore not omitted.

Figure 3.11 also illustrates another important aspect of the model, namely error handling. One transition in the figure handles a negative acknowledgement indicating failure to establish a connection. The third transition in the figure illustrates a design choice for handling both user-entered interruptions and network failures, which were general events for most states in the model.

The solution for handling errors was to add internal transitions or self-transitions for such events in the top-level state machine. Self-transitions for user interruptions and network failures were added to the Authentication state (which included an internal state machine), as illustrated in Figure 3.12.

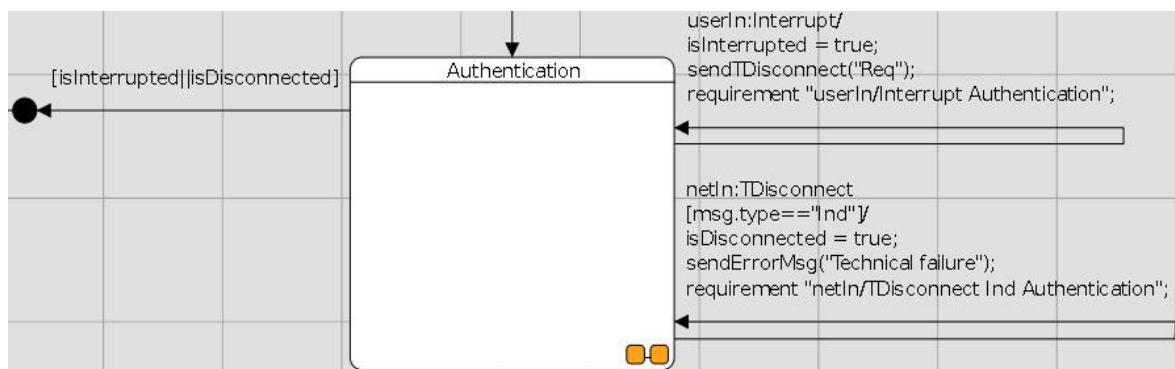


Figure 3.12: Experiment 1: Authentication in top-level state machine

These transitions restarted the internal state machine, while including appropriate actions. The defined self-transitions also set Boolean data members that were used to terminate the internal state machine, as illustrated by the Boolean data members in Figure 3.11. Since the internal state machine was restarted the solution required two transitions (illustrated by the transitions using Boolean data members in Figure 3.11 and Figure 3.12) to terminate execution thread of the complete model.

However, this solution using self-transitions only applied to the Authentication state and its internal state machine because in that case all states required the same error handling. For the Withdrawal state machine not all states required the same error handling, therefore internal transitions for each state was applied. Furthermore, the top-level state machine also included states not regarded with internal state machines, to which the same solution using internal transitions applied. This is illustrated in Figure 3.13 below.

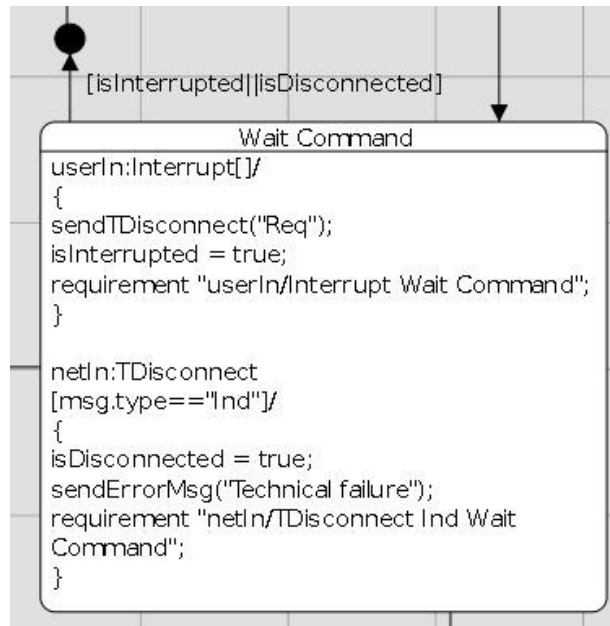


Figure 3.13: Experiment 1: State of top-level state machine

The use of internal transitions is illustrated in Figure 3.15. The transition for terminating the execution thread only used a guard (see section 3.2.1) including Boolean data members. The solution of internal transitions decreased the number of transition lines in the graphical notation by only using one transition for error handling from each state. Since all transitions of the top-level state machine ended in the same final state this was a desired outcome.

Test generation

Figure 3.11 also includes the requirement keyword (see Appendix C.2.1). This is a QML statement which also may be specified in the textual notation. This statement marks a point in the model that can be used as a testing goal for Qtronic, and the string specified is the name of the requirement and has to be unique. The specified string was named according to the input, and grouped as network input or user input using the “/”-character (Qtronic creates a tree structure for the requirements in the test design configuration based on this character). The test design configuration (see Appendix C.2.1) of this experiment was set to cover these requirement statements, which covered all the possible inputs for both interfaces.

The test generation options (see Appendix C.2.2) were set to only finalized runs and a lookahead depth of 2. The implication of the only finalized runs option is that Qtronic only produces test cases ending in a final state of the top-level state machine. The lookahead depth was set to the lowest possible value resulting in complete coverage, i.e. that all requirement statements were covered. The minimal lookahead value for this model resulting in full coverage was 2, the second lowest value possible.

Test harness and test execution environment implementation

The abstract test cases generated in Qtronic were then rendered to executable test scripts in TCL. The test cases included in the test suite file were as previously mentioned (see section 3.3.3) complete, but the test harness needed to be implemented. Since this was the first experiment, the test execution environment had to be implemented in TCL as well. The test harness and the test execution were implemented in parallel. The test execution environment

was implemented as described in section 3.2.4 using socket communication to interact with the test object. The test harness procedures used to define the test cases were implemented to either send or receive data on the socket channel, as described and exemplified in section 3.3.3.

```
proc netInTData { type header } { \
    global sockChan
    set msg "T-Data $type $header"
    send $sockChan $msg
}
```

This is a sample procedure of the test harness implemented for this experiment. This particular procedure sends a string of a certain structure to the SUT over the socket channel.

Test execution

After implementing the test harness procedures and the test execution environment the test scripts were executed against the SUT (see section 3.3.4). The initial test execution resulted in a number of failures. The failures were both due to errors in the model and in the test object. The failures were either due to mismatches in the comparison between the expected and the actual output, or due to deadlocks, i.e. the sequence of ingoing and outgoing messages was not consistent for the model and the SUT. The latter resulted in that the test execution halted when both the test script and the SUT expected to receive data on the socket channel. This scenario occurred when the number of actions for a given transition differed between the test object and the model. The errors were analyzed using the generated test execution log file, which indicated where errors first occurred. The errors were then corrected by manually inspecting the model, the test harness and the test object.

Results

Table 3.1 below includes the results of this initial experiment. Some results are derived from Qtronic and some are indications on the work effort required to for certain tasks in the Qtronic testing process.

Modeling time	2 days
Test generation time	13 seconds
Test design configuration coverage	100%
Number of generated test cases	25
Time to implement test harness	2 days
Lines of code: Test suite	2860
Number of test harness procedures	18
Lines of code: Test harness	99
Average: LOC / Harness procedure	5.5
Lines of code: Test execution environment	73

Table 3.1: Results experiment 1

The modeling time was estimated to be two full working days, where one working day was approximately 8 hours. For this experiment one of the Conformiq instructors assisted with modeling of the system and the model was completed during the two days he stayed.

The test generation time, the test design configuration coverage and the number of test cases are derived from Qtronic. The test coverage was inspected by analyzing the generated test cases and confirming the coverage reported by Qtronic.

The time to implement the test harness procedures is an indication of the work effort required to successfully test and communicate with the SUT. However, the time to implement the test harness for this experiment also included the time to implement the test execution environment in TCL. The two were implemented in parallel, thus the estimated time includes both tasks. The lines of code metric for the test suite is provided as an indication of the size of the test script generated and defined by Qtronic. The lines of code measures for the test harness are provided as an indication of the work effort required for the script implementation that has to be done by the tester. Furthermore, the average of lines of code for each test harness procedure gives an indication of the work effort required for each procedure to be able to test against the SUT.

The experiment and the results are analyzed and evaluated in section 4.2.

3.5.2 Experiment 2

The second experiment used an extended specification (see Appendix C.1.2) compared to the initial specification (see Appendix C.1.1). This experiment followed the same methodology and process as the first experiment and the introductory example (see section 3.3). However, compared to the initial experiment, this experiment made use of existing artifacts: the test object, the QML model and the test harness from the first experiment. Furthermore, when starting this experiment the test execution environment was implemented (see Experiment 1 in section 3.5.1).

Goal

The goal of this experiment was to add requirements and to extend the specification to see how the implication of an extended model propagates through the different tasks involved in the Qtronic testing process compared to the initial model.

Implementation

When starting this experiment the test object from the first experiment was reused. The existing test object was extended according to the added requirements and the new specification (see Appendix C.1.2), using the same design pattern as for the initial version. Furthermore, the test object in this experiment was extended to handle parameter logic for different messages. The parameter logic is described in more detail for the modeling of this experiment.

Modeling

The QML model of the first experiment was extended according to the new specification (see Appendix C.1.2). The PMC top-level state machine was extended with two new states (Deposit and Transfer), symbolizing two added functions for the user interface, namely account transactions and account deposits. These two states included internal state logic in terms of internal state machine, which were defined separately according to the specification (see Appendix C.1.2).

When extending the graphical QML model, the top-level state machine was difficult to extend, since the state machine only used one final state. All paths, including error handling for all states, ended in that final state. Thus, to simplify the top-level state machine more final states were introduced. This simplified the drawing within Qtronic Modeler and made the model more readable. Final states were introduced for all error handling transitions, both in the top-level state machine as well as in the internal state machines. This practice was implemented in the internal state machines, although not really necessary, to use the same methodology for all state machines. For each state of the model one final state was added to handle unexpected errors, such as user entered interruptions and network failures, as well as a transition from the state to the final state. Furthermore, for states expecting to receive acknowledgements one additional final state was added along with a transition for handling a negative acknowledgement. The error handling as modeled in this experiment is illustrated in Figure 3.14 below.

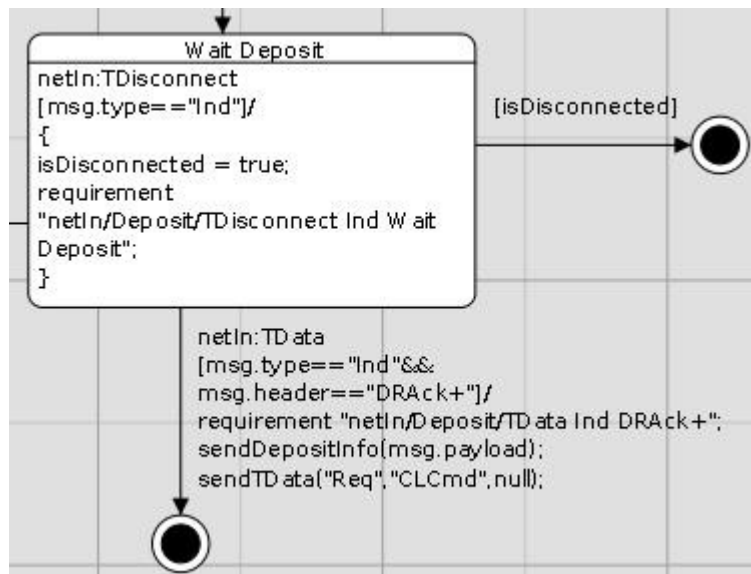


Figure 3.14: Experiment 2: Error handling

Figure 3.14 illustrates the introduction of final states for the error handling of each state. In this particular state a user entered interruption was not a valid since the deposit already had been made and the protocol module was waiting for the server to confirm the deposit.

All state machines except the Authentication state machine required different error handling for the states contained in a particular state machine. Since the Authentication state machine was the only exception the same error handling methodology (as described and exemplified above) was applied to that state machine (compare to initial experiment as described in section 3.5.1). Hence the self-transitions used in the initial experiment were removed. Instead internal transitions were applied for unexpected events, such as user entered interruptions and network failures, as illustrated in Figure 3.14.

Since the model was extended new message types were defined and the system block was updated accordingly. All new message types were messages of the user application interface of the model since generic network message types already were defined. New class methods for the top-level state machine were also defined for the new message types, as discussed in Experiment 1 (see section 3.5.1).

Compared to the first experiment, the model of this experiment included message parameters, or record data members, such as payload for network messages, card number and pin code for messages sent of the user application interface. This is exemplified in the code below in the *TData* record.

```

record TData {
    public String type;
    public String header;
    public int[] payload;
}
  
```

Compared to the first experiment (see section 3.5.1) the network messages may include any number of integer values. This particular message type is defined in this way since it is a generic construct that is reused for a significant number of transitions. Hence, the number of

data values included in a message varies depending on the header, i.e. what command (for outgoing actions) or acknowledgement (incoming events) the message includes.

```
class PMC extends StateMachine {
    // ...
    public void sendTData(String type, String header, int[] payload){
        TData r;
        r.type = type;
        r.header = header;
        r.payload = payload;
        netOut.send(r);
    }
    // ...
}
```

The QML procedure defined above was used in the graphical notation to send network data. The payload, which is an integer array, is initialized in the graphical notation. The reason for this design is that the scope of an incoming event in QML only encompasses that particular transition string (see Qtronic Modeler in section 3.2.1). In many cases parameters of events are passed on to a following action message of that transition.

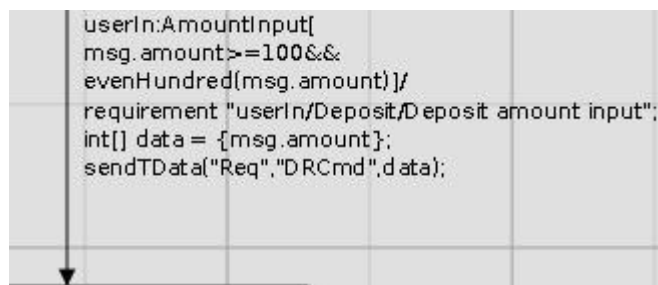


Figure 3.15: Experiment 2: QML transition example

Figure 3.15 illustrates this logic. In this particular case the event is that the user has provided the deposit amount. The provided deposit amount is then included in the network request sent to the server, which processes this request. The action uses the class method *sendTData* as defined in the sample code above. The initialization of the payload array could have been specified in a class method of the state machine. However, that would have resulted in a class method for each transition including payload data since the number of event parameters varies. Furthermore, the payload is not set for all transitions and in the case when it is not, the payload is initialized to the value null by providing the null value as the third parameter to the class method. If the array was not initialized to null in such cases Qtronic would still automatically do this during test generation. The reason for the null initializations in the model is to use one generic class method, regardless of the payload is used or not.

The model of this experiment used transition guards (see section) to control values for message parameters during test generation. The transition guard in Figure 3.15 describes that the deposit amount has to be equal to or greater than hundred (according to the Swedish ATMs and currency) and that the amount has to be of even hundreds. Furthermore, require statements in the code were also used to control test generation data, as illustrated in the sample code below.

```

public void sendTransactionInfo(int[] payload) {
    require payload!=null;
    require payload.length==2;
    require payload[0]==enteredAccount;
    require payload[1]==enteredAmount;

    TransactionInfo r;
    r.account = account;
    r.amount = amount;
    userOut.send(r);
}

```

The require statements were used to control the size of the generic payload array and to control the value generations for the array. In this particular case a transaction request has been acknowledged by the server and the require statements are necessary to ensure that the acknowledged transaction information corresponds to the account and amount entered by the user. These statements are used to model dependencies between different incoming messages (entered account, entered amount and acknowledged transaction).

Test generation

As for the first experiment, the test design configuration (see Appendix C.2.1) was set to cover requirement statements in the model. In this experiment the requirements were further grouped, as specified in Figure 3.15. The requirements, corresponding to events, were now not only grouped by which interface the input originated from. At the highest level they were still grouped according to which interface they originated from. However, the second level of grouping was in which state machine they were specified. In this way it was easier to track the requirements and control the test generation for particular parts of the model.

The test generation options used (described in Appendix C.2.2) were set to use only finalized runs and a lookahead depth of 2. This value, as in Experiment 1, was set to the minimal value which resulting in full coverage i.e. covered all requirement statements in the model.

Test harness implementation

The test harness implemented for the first experiment was reused initially. However, this model included message parameters and payload data. Therefore the signatures for a number of harness procedures were different, since data members were added to the appropriate message types. Since the test harness procedures are generated from the system block and the message definitions, as described in the introductory example (see section 3.3.3), several of the procedures generated included additional parameters compared to the test harness of the first experiment. Nevertheless, most of the existing test harness procedures were reused with small extensions. However, a small number of procedures required more implementation.

```

proc netInTData { type header payload } { \
    global sockChan
    set msg "T-Data $type $header"
    if { $payload != "{}" } {
        set l [split $payload ,]
        for {set i 0} {$i < [llength $l]} {incr i} {
            set msg "$msg / [lindex $l $i]"
        }
    }
}

```

```

    }
}
send $sockChan $msg
}

```

The sample procedure above is one of the procedures that required more implementation compared to the equivalent procedure in the initial experiment (see test harness implementation of Experiment 1 in section 3.5.1). The payload array used in the QML model was not rendered to a TCL array but to a TCL string. This string had to be parsed to construct the format of the data to be sent on the socket channel. The pipe character was used as a convention to indicate that data parameters were provided.

Moreover, a few new procedures were generated from this model according to the new message types defined in the model. As discussed for the modeling of this experiment, the new message types were all defined for the user application interface.

Test execution

The initial test execution resulted in a number of failures. As for Experiment 1, the failures were either due to mismatches in the comparison between the expected and the actual output or to inconsistencies between the model and the test object. The latter failures occurred when the number of actions for a given event differed between the test object and the model. This resulted in that both the test script and the test object expected to receive messages on the socket channel. The errors were corrected with the help of the log file and manual inspection of both the model and the code. The log file indicated the last successful execution. Thus it indicated where to inspect the model, the test harness and the code of the test object.

Results

Table 3.1 below includes the results of this initial experiment. Some results are derived from Qtronic and some are indications on the work effort required to for certain tasks in the Qtronic testing process.

Modeling time	1 day
Test generation time	2 min 34 sec
Test design configuration coverage	100%
Number of generated test cases	52
Time to implement test harness	1 hour
Lines of code: Test suite	6278
Number of test harness procedures	26
Lines of code: Test harness	155
Average: LOC / Harness procedure	~5.96
Lines of code: Test execution environment	73

Table 3.2: Results experiment 2

The modeling time was estimated to be one working day, or 8 hours. The modeling may have been performed in a shorter period of time since the model of the first experiment was reused. However, while creating the model different design choices within QML were evaluated before completing the model. Hence the modeling task required more time.

The test generation time, the test design configuration coverage and the number of test cases are derived from Qtronic.

The implementation the test harness procedures required about 4 hours of work. Most of the time was spent initially before figuring out exactly what the TCL scripting backend actually rendered. An example of this is that the scripting backend rendered the QML integer array as a string and not as a TCL array.

The experiment and the results are analyzed and evaluated in section 4.3.

3.5.3 Experiment 3

The third experiment used a changed and modified specification (see Appendix C.1.3) compared to the specification used in the second experiment (see Appendix C.1.2). This experiment followed the same methodology and process as the first two experiments and the introductory example (see section 3.3). However, in this experiment the existing specification was modified by adding a general biometric authentication, required to complete any account requests, which meant that the internal state machines had to be modified. This resulted in the desire to apply a QML structure that could be reused in several state machines.

Goal

The goal of this experiment was to change the requirements and to modify the specification to see how the implication of modifications in the existing model propagates through the different tasks involved in the Qtronic testing process compared to the model of the second experiment.

Implementation

In this experiment the test object from the second experiment was reused. The existing test object was modified according to the added requirements and the new specification (see Appendix C.1.3), using the same design pattern as for the other two versions of the test object. The test object in this experiment required a successful biometric authentication before completing account requests.

Modeling

The QML model of the second experiment was reused and modified according to the new specification (see Appendix C.1.3) to require a successful biometric authentication to complete account requests.

This model followed the same design as the model of the second experiment. The top-level QML state machine was not modified in either model. However, the internal state machines needed to be modified to support the biometric authentication. Since the same logic was used for all account requests, namely withdrawal, transactions, deposits and balance requests, the aim was to design a reusable structure and not to repeat the same logic in all internal state machines.

This was achieved by creating a new and separate QML model, including the biometric authentication logic. This model was a simple state machine including only three states (as specified in Appendix C.1.3). This state machine was then used in the internal state machines. Furthermore, an internal state machine was created for the state Balance (as described in Appendix C.1.3) to keep the top-level state machine simple and instead encapsulate the biometric authentication in that internal state machine. Thus all main functions of the protocol module, namely authentication, withdrawal, transfer, deposit, balance requests and biometric authentication, were modeled as internal state machines.

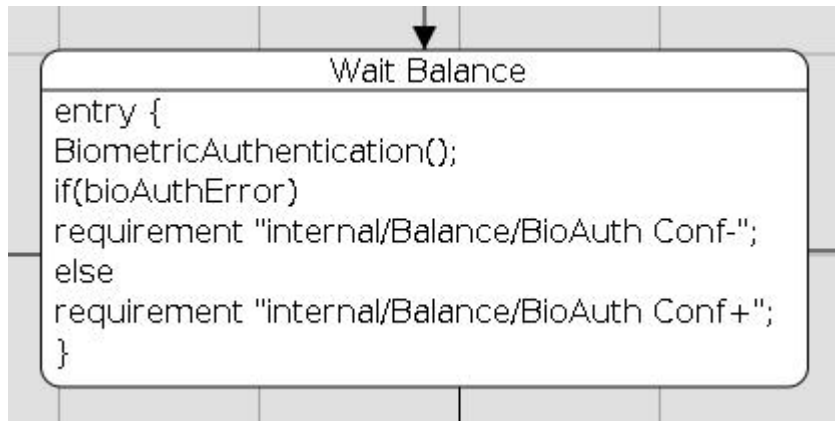


Figure 3.16: Experiment 3: Biometric authentication

Figure 3.16 illustrates the use of the biometric authentication state machine. This figure is a screenshot from the internal state machine for balance requests. The procedure in Figure 3.16, *BiometricAuthentication*, starts a new thread of the biometric authentication state machine, which is a separate QML model. The procedure, which is defined for the main QML model, then waits to receive an internal message from the biometric authentication thread. Thus two threads within Qtronic project, namely the instance of the protocol module model and the instance of the biometric authentication model, communicate internally. If the received internal message indicates successful authentication the execution continues. In the other case the balance state machine will go to a final state and then the top-level state machine will terminate in the same manner. The same logic applies for the internal state machines for withdrawal, transfer and deposit. This is described in more detail in the specification (see Appendix C.1.3).

The QML model was designed as the model of the second experiment, including logic specified as in the specification, with the modifications as described in this section. A typical transition of this model is illustrated in Figure 3.15 (see section 3.5.2), where the payload is initialized. Thus this model and the model of the second experiment were largely similar, with the exception of the biometric authentication process modeled in this experiment.

Test generation

As for the first two experiments, the test design configuration (see Appendix C.2.1) was set to cover requirement statements in the model. The requirements in this experiment were grouped as described for the second experiment (see section 3.5.2). Since the requirement statements were specified based on incoming messages, or events, the requirements were exactly the same for the two versions of the model.

The test generation options (described in Appendix C.2.2) were set to use only finalized runs and a lookahead depth of 2. This value, as in the first and second experiments, was set to the minimal value which resulted in full coverage.

Test harness implementation

The test harness reused the entire test harness of the second experiment. However, the biometric authentication model rendered two additional procedures corresponding to messages for the user application interface (*userIn* and *userOut* in the model), which required implementation. The messages were request and input of biometric information.

Test execution and analysis

The test execution of this experiment resulted in a small number of failures. These failures were caused by errors in the test object and were deadlocks (as discussed for the initial experiment). Deadlocks in this context means that the QML model and the test object were inconsistent which resulted in an asynchronous communication between the test scripts and the test object over the socket channel. The implementation of the test object was missing some error handling for the biometric authentication functionality. The errors were detected while comparing the QML model and test object implementation. The test harness contained no errors since the greater part of the procedures were reused from experiment 2 and hence tested in that experiment.

Results

Table 3.3 below includes the results of this experiment. Some results are derived from Qtronic and some are indications on the work effort required to for certain tasks in the Qtronic testing process.

Modeling time	4 hours
Test generation time	3 min 11 sec
Test design configuration coverage	100%
Number of generated test cases	56
Time to implement test harness	10 min
Lines of code: Test suite	7540
Number of test harness procedures	28
Lines of code: Test harness	165
Average: LOC / Harness procedure	~5.89
Lines of code: Test execution environment	73

Table 3.3: Results experiment 3

The modeling time of this experiment was estimated to one working day, or about 8 hours. The changed requirements (as specified in Appendix C.1.3) did not require much time in terms of implementation. However, more time was spent figuring out how to model this general functionality (the biometric authentication) that was added to several internal state machines (as discussed in the modeling paragraph of this experiment). Hence, most of the modeling time was spent on exploring constructs and therefore possibilities of QML.

The time to implement the test harness is estimated to only 10 minutes. All test harness procedures of this experiment except two were reused from experiment 2.

The experiment and the results are analyzed and evaluated in section 4.4.

3.5.4 Experiment 4

The fourth and final experiment modified the model of the third experiment, still using the same specification (see Appendix C.1.3). This experiment did not follow the same methodology as the second and third experiment, in which the specification was extended. This experiment used the same specification and the same version of the SUT as the third experiment but involved the evaluation of testing the SUT by extracting some logic from the QML model and implementing that logic in the test harness.

Model Development

The model of this experiment was created using the model of the third experiment (see section 3.5.1). The model of the third experiment was modified by removing dependencies between parameters of different messages in the model.

In the model of the third experiment parameters of an outgoing message (action) were in a number of cases dependent on parameters of the incoming message (event). For example, given an incoming message to the protocol module from the user application interface indicating an entered withdrawal amount the protocol module sends a network message to the server. The parameter of the network message sent to the server is the payload, which in this particular scenario includes the withdrawal amount as provided by the parameter for the event. These modeled dependencies between incoming messages and outgoing messages were removed from the action part of transitions in the model.

Furthermore, the models of the second and third experiments included require statements (see section 3.5.2) that were used to describe the dependencies between parameters of different incoming messages. An example is the dependency between an incoming message on the user application interface indicating an entered withdrawal amount and an incoming message on the network interface indicating the acknowledgement of a withdrawal. In this case require statements were used for the incoming network message to describe that the acknowledged withdrawal amount (included in the payload) had to be the same as the withdrawal amount entered by the user. Require statements describing such dependencies were also removed from the model.

```
userIn:AmountInput[
msg.Amount>=100&&
evenHundred(msg.amount)]/
requirement "userIn/Deposit/Deposit amount input";
sendTData("Req","DRCmd");
```

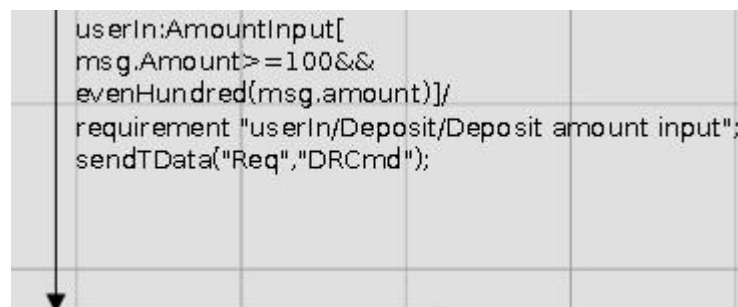


Figure 3.17: Experiment 4: QML model transition

Comparing Figure 3.17 to Figure 3.15 illustrates the difference between models of this experiment and the third experiment. Although removing logic from the model in this experiment, the intention was that Qtronic would fulfill the same purposes as for the third experiment and generate the same test cases. The removed logic describing message dependencies is further illustrated in the sample code below.

```
public void sendTransactionInfo() {
    TransactionInfo r;
    userOut.send(r);
}
```

This sample code may be compared to the sample code of the second experiment (see section 3.5.2). In this model the require statements describing dependencies between user entered input and network messages from the server were removed. In this experiment the message parameters were still included, but the logic describing the dependencies between parameters of different messages was removed. However, the overall logic is the same for both models.

Test generation

As for the first two experiments, the test design configuration (see Appendix C.2.1) was set to cover requirement statements in the model. The requirements in this experiment were grouped as described for the third experiment (see section 3.5.3), which is illustrated in Figure 3.17. Since the requirement statements were specified based on incoming messages, or events, the requirements were exactly the same for the model of this experiment as for the model of the third experiment.

The test generation options (described in Appendix C.2.2) were set to use only finalized runs and a lookahead depth of 2. This value, as for the previous experiments, was set to the minimal value which resulting in full coverage.

Test harness implementation

The test harness corresponding to modified QML model required more implementation. Most of the procedures were similar to previous implementations, following the same pattern as receiving or sending procedures. However, especially generic message types and the corresponding test harness procedure required more implementation to describe the logic and dependencies between parameters of different messages. The most generic message type was *TData*, which was the message primarily used for the network interface. The two test harness procedures, one corresponding receipt of network data and one for sending network data, required the most implementation.

```

proc netInTData { type header payload } { \
  global sockChan
  global amount
  global account
  global balance
  global seed

  set data ""

  if { $header == "DRAck+" || $header == "DRAck-" } {
    set data "$amount"
  } elseif { $header == "TRAck+" } {
    set data "$account | $amount"
  } elseif { $header == "WRAck+" } {
    set data "$amount"
  } elseif { $header == "Seed" } {
    set data "$seed"
  }

  if { $data != "" } {
    set msg "T-Data $type $header / $data "
  } else {
    set msg "T-Data $type $header"
  }
  send $sockChan $msg
}

```

This is a sample procedure of test harness in this experiment. This sample procedure may be compared to the test harness sample code for the corresponding procedure of the second experiment, which was reused in the third experiment (see section 3.5.2).

Test execution and analysis

The test execution of this experiment resulted in a small number of failures. The test object had been updated in the test execution of the third experiment, thus not containing errors in this test execution. The QML model was also not containing any errors. Hence, the errors were found in the implementation of the test harness. The failures of this test execution were output mismatches, and not deadlocks as for the test execution of the third experiment. The output mismatches were indicated in the generated log file and due to differences between the expected and actual output format of the SUT. Since the log file indicated that the data values of messages and not the message types differed for the expected and the actual output, conclusions were made that the test harness contained errors. The errors were then removed by inspecting the test harness implementation.

Results

The results of the fourth experiment are summarized in Table 3.4 below.

Modeling time	2 hours
Test generation time	3 min 6 sec
Test design configuration coverage	100%
Number of generated test cases	56
Time to implement test harness	2 hours
Lines of code: Test suite	7476
Number of test harness procedures	28
Lines of code: Test harness	229
Average: LOC / Harness procedure	~8.18
Lines of code: Test execution environment	73

Table 3.4: Results experiment 4

When creating the model for this experiment the model of the third experiment was reused and slightly simplified. Since the model included a number of state machines it still required some time to verify that the changes were correct and still included the same overall functionality as the model of the third experiment.

As discussed in the test harness implementation paragraph of this experiment description, test harness implementation required significantly more time compared to the third experiment. However, many of the procedures were still very similar to the test harness of the third experiment and were quickly implemented. A few procedures, corresponding to generic message types, required more implementation.

The test generation times, the test design configuration coverage and the number of generated test cases for the experiment are all derived from Qtronic.

The experiment and the results are analyzed and evaluated in section 4.5.

3.6 Summary

In this chapter the project work was described. The chapter started by briefly describing the purpose of the thesis work, which was to evaluate the concept of MBT by using the MBT tool Qtronic and incrementally develop the test object.

Section 3.2 defined and described the test system of the project, including Qtronic, the test object (the SUT), the generated and implemented test scripts involved in the test execution as well as the test execution environment. The description of Qtronic gave an introduction to the tool. This introduction included the fundamental tasks (modeling, test generation, script generation) and features (QML, Qtronic Modeler, Qtronic projects) of Qtronic. The test object description addressed the SUT of the project, which was a simplified model of the client-side protocol module of an ATM client-server system. The test object was implemented according to the specifications (see Appendix C.1) and the state design pattern [15] in Java. Qtronic generated two TCL scripts of interest: the test suite file and the test harness file. The test suite script included all test cases and was completely generated by Qtronic. The test harness included procedure declarations in TCL but required implementation by the tester. Finally, section 3.2 described the implemented test execution environment, i.e. how the test scripts and the SUT communicated. The test execution environment involved both TCL and Java socket implementation.

In section 3.3 an introductory example to the project work was given. This example was given to describe the way of working and the process involved when testing with Qtronic. This example also described how the different tasks were related. The model used in the example was a subset of the complete model; hence the generated test cases were still valid for the SUT. The example only modeled the functionality for requesting account balance information of an ATM.

The last section of the chapter included the four experiments of the thesis. These experiments were performed in chronological order, that is, they were based on the incremental development of the test object. The initial experiment was to create a simple model of the SUT according to the initial specification (see Appendix C.1.1), to successfully execute test cases and to verify that the test execution environment worked as intended. The goal of the second experiment was to investigate how added requirements and functionality propagated through the different tasks of the Qtronic testing process. The goal of the third experiment was to investigate the implication of changed requirements in the specification and how they propagated through the model. The goal of the fourth and final experiment was to investigate the implications of moving logic from the QML model to the test harness. The first experiment was compared to the second experiment, the second experiment was compared to the third experiment and the third experiment was compared to the fourth experiment.

4 Results and Evaluation

This chapter includes the analysis of the four experiments performed as well as the evaluation of the project as a whole. The analysis of the project work is a qualitative analysis, including each individual experiment as well as the project at large, Qtronic and Qtronic Modeler.

Referring to the discussion of design factors in modelling included in chapter 3, these aspects are briefly repeated. Since modelling is the most important and the crucial task of the MBT approach the design aspects needs to be emphasized. The design aspects will be a point of discussion in the analysis of the experiments.

The chapter includes analysis of each experiment performed in this project. The experiments are analyzed individually but also compared to the previous experiment since the project was based on incremental development of the test object. The general structure of the analysis is based on the working process of the project, thus including model development, test generation, test harness implementation, test execution and finally results. This structure applies to the analysis of each experiment. The last part of the chapter is a project analysis. This analysis is more general compared to the experiment analysis and focuses on the project at a higher level. Furthermore, the project analysis includes evaluations of the MBT tool Qtronic and the modelling tool Qtronic Modeler.

4.1 Design Factors in Modeling

As discussed in section 3.4, design factors are important when modeling and impact on the quality of the model. Since the model is the most important artifact in the model-based testing (MBT) approach such aspects need to be emphasized and discussed. Thus design factors will be included in the analysis of the experiments of this project.

The particular design factors that will be discussed in this analysis are readability, extendibility and maintainability. The reason for considering readability is the nature of the model, meaning that the model has the potential to serve as an effective means of communications. This readability is an important design consideration so that testers can understand the model and for ensuring that models should be constructed according to established design guidelines.

Extendibility is an important design factor since systems often are developed incrementally. For complex systems this design factor may be more important than for systems resulting in smaller models, but the models should support incremental development. Hence it is important to considering the extendibility of the model, since it is very likely that the system and consequently the model will be extended.

The third design factor, maintainability is also an important design factor. As systems are developed incrementally and evolved through time, maintenance is often an important and obligatory part of projects. Complex systems often include a great deal of maintenance, thus maintainability is an important consideration when designing models for testing.

4.2 Experiment 1

As described in section 3.5.1, the goal of the initial experiment was to apply MBT to an existing test object and to execute the rendered and implemented test scripts against that test object (the SUT). To achieve this goal the different tasks in the Qtronic testing process needed to be successfully completed. The main tasks included the creation of a QML model, the test harness implementation and the test execution environment implementation. The implementation of the test execution environment was only a task of this initial experiment. When analyzing this experiment it is important to consider the learning phase of applying Qtronic for performing MBT, primarily including creation of the QML model and the test harness implementation.

Measures	Experiment 1
Modeling time	2 days
Test generation time	13 seconds
Test design configuration coverage	100%
Number of generated test cases	25
Time to implement test harness	2 days
Lines of code: Test suite	2860
Number of test harness procedures	18
Lines of code: Test harness	99
Average: LOC / Harness procedure	5.5
Lines of code: Test execution environment	73

Table 4.1: Results experiment 1

The analysis in this experiment focuses on the learning phase (“Modeling time” + “Time to implement test harness”) when applying MBT using Qtronic.

4.2.1 Model Development

The first testing phase of the thesis work was to create a QML model corresponding to the initial test object (implemented using the specification in Appendix C.1.1). The creation of the first Qtronic project and the first QML model implied a learning phase in terms of the QML textual notation, the QML graphical notation, design practices as well as tool specific features.

Basic system

The QML model of this experiment was created using the initial specification (see Appendix C.1.1), as described in the experiment description (see section 3.5.1). The specification was a UML diagram (see Appendix C.1.1) and did not include error handling. Hence the first version of the model depicted the basic system without error handling. The modeling of the basic system was a straightforward task since the specification was a UML diagram and the Qtronic Modeler made use of UML notation. Hence, the UML diagrams and the QML model were initially very similar, although the transition logic of the QML model was defined using a greater level of detail. One state in the UML diagram resulted in one state in the QML model. The modeling of the basic system was therefore completed in a relatively short period of time (about 2 hours - a part of the time given under “Modeling time” above i.e. 2 days).

At this point design factors such as readability, extendibility and maintainability were considered when modeling. The functionality of the protocol module was encapsulated and grouped (using internal state machines as described in section 3.5.1). The goal was to group states, or functionality, to create two hierarchies of the model to simplify the top-level state machine and increase the level of abstraction. For example, all states in the authentication process of an ATM request were grouped in a separate state machine. The model consisted of the top-level state machine and two internal state machines, namely Authentication and Withdrawal. The reason for introducing these internal state machines was different design factors, namely readability, extendibility and maintainability of the model. By encapsulating logic within separate state machines, bound to states of the top-level state machine (i.e. internal state machines), readability was enhanced since the model at this point included several smaller state machines instead of one state machine containing all the logic. For example, when inspecting the withdrawal process there was no need to know specific details of the authentication process, just that authentication was required for a withdrawal request. However, the balance functionality was not encapsulated in a state machine since it only included one state specific for that request.

Moreover, the encapsulation had implications for extendibility of the model since one added function (user entered ATM request) to the protocol module would require only one additional state in the top-level state machine as well as a new distinct state machine bound to the new state (internal logic of the new state). Hence, extended or changed requirements would require small adjustments in the top-level state machine of the model. The focus in such cases would be on the design of the state machine describing the internal state logic (i.e. internal state machines), thus yielding a higher level of abstraction. The task of creating the basic system was straightforward since only basic structures of the QML textual notation (procedures) and the graphical notation (transitions and states) were used.

Error handling

The next step of the modeling task was to add error handling for the protocol module. The encapsulation of logic described for the basic system had implications for the error handling in the model. The error handling in terms of negative acknowledgements had to be modeled at a state-level since they were specific for particular states. However, other error messages were general for most states. When the error handling first was added to the model the number of transitions from each state increased significantly, having an impact on readability and expandability. This was partially an issue with the drawing tool (Qtronic Modeler) but also with the design in general. The issues and experiences of Qtronic Modeler are discussed in the project analysis (see section 4.7.3).

The model of this experiment included two primary encapsulations of functionality, namely for withdrawal and authentication (as described in section 3.5.1). The states of the authentication process required the same error handling. Thus, the error handling for the authentication process was applied at a higher level and in a uniform way, which simplified the authentication state machine. However, the states encapsulated within the withdrawal state machine required different error handling. Hence the error handling had to be handled at a state-level within that particular state machine. The encapsulation for withdrawal could have omitted some states to achieve the same solution as for the authentication process. However, this would have been done in contrast to the design factors as previously discussed and only in an effort to simplify error handling and only to decrease the number of transitions connected with error handling (see section 3.5.1 for a detailed discussion regarding the error

handling solution). Hence the error handling was handled at a state-level within the withdrawal state machine.

The modeling of the error handling required more time than the creation of the basic system. When error handling was added to the basic system the number of transitions increased significantly. Therefore the Qtronic User Manual [8] was consulted to find a suitable solution for decreasing the number of transitions, thus impacting readability and extendibility. Investigating the QML structures, thereby applying and test them, required more time.

4.2.2 Test generation

The test generation was based on the requirement statements, as stated in section 3.5.1. The requirements were stated for all incoming messages, both from the user application interface and the network interface. In this way they were grouped by interface which became a useful property for test generation and test execution when errors were analyzed.

The other test design configuration parameters (see Appendix C.2.1) were not tested systematically. The reason for this was that a systematic approach of evaluating test generation times and the test design configuration parameters were not the purpose and goal of the experiment or the project. However, when exploring the options and features in Qtronic the different parameters were used for test generation to see how they worked. Requirements were chosen for simplicity as well as for tracking the relationship between the generated test cases and the events of the system.

A particular difficulty was encountered during the test generation. The error handling for the authentication state machine was handled in a uniform way. However, the test generation only included one test case for each error (user entered interruption and network failure) of this state machine. For example, interruption was only tested for one of the states (the initial state) within the internal state machine. This was not an important issue, but it was desired to generate all possible inputs for every state of the SUT.

It was later learned that this could have been achieved by using a different test design configuration, namely dynamic coverage and all paths-states (see Appendix C.2.1), which aims at covering every sequence of states. The result was that the errors mentioned were not tested for all states within the authentication process but only for one state. Specifically, user entered interruption and network failure were only tested for the initial state of the Authentication state machine Using the alternative test generation parameter resulted in six additional test cases since the Authentication state machine included four states and each state included two possible errors, while one state already was covered in the initial test generation.

4.2.3 Test harness and test execution environment implementation

When the QML model was completed the next task was to execute the test cases against the SUT. The test harness implementation however required a test execution environment, i.e. a means of sending input and retrieving output from the SUT. After discussions with the Conformiq instructor it was decided to use a socket channel to achieve this, where the SUT acted as a socket server. General procedures for sending and receiving data over the socket were implemented both for the SUT and for the test scripts in TCL. The most important consideration was to ensure that the communication was synchronized (assuming a correct

model and SUT). The sending procedures simply sent data to the socket channel while the receiving procedures waited until something was written to the socket channel.

When the test execution environment was implemented and tested, the test harness implementation was straight-forward. All messages modeled as incoming in the model became sending test harness procedures and correspondingly all messages modeled as outgoing messages became receiving test harness procedures. The test harness procedures only included implementation for the socket communication. This is illustrated in the sample code below.

```
proc userInAmountInput { } { \  
    global sockChan  
    set msg "Amount input"  
    send $sockChan $msg  
}
```

The log functionality and the verdict functionality (see section 3.2.4), i.e. comparing expected and actual output, were implemented for the general sending and receiving procedures. Thus the challenge of this task was to define and implement the test execution environment. If such an execution environment had been in place and defined from the start the task of implementing the test harness would have been a small and straightforward task.

The initial expectation was that the test harness would be a reflection of the system block (section 3.3.1) and the message definitions of the model as well as the test execution environment. After selecting the test execution environment the expectation was that the test harness procedures would simply send or expecting to receive strings on the socket channel based on the procedure name (as illustrated in the sample code above). The expectations about the test harness implementation were fulfilled, and all the test harness procedures either sent or received data on the socket channel.

4.2.4 Test execution

Given the implemented log functionality, where all input, expected output and actual output was written to a log file, the test execution analysis was not a challenging task. Depending on the error indicated in the log file different measures were taken. In the simple case the failures were due to errors in terms of string mismatches. In the worst case the errors were due to deadlocks, i.e. the model and the SUT were not consistent which resulted in both the SUT and the test script expecting to receive data on the socket channel. A common error at first was that a transition was missing in the test object. However, the log file indicated the failing events and thus indicating where to inspect both in the model and the SUT implementation.

The test execution and the generated log file were what would be expected when performing black-box testing, including input, expected output and actual output. The expected output was not explicitly generated, as may have been expected, but was implicitly known within each receiving test harness procedure along with potential parameters of that procedure. The expectation from reading about MBT was that this was completely automated from the model. However, the solution, or implicit generation, of expected output from using Qtronic was sufficient. In this way it is up to the tester to handle this information. The implication is that the implementation of the expected output in the test harness depends on the abstraction level of the model.

4.2.5 Results and conclusions: experiment 1

The results of this experiment are given in Table 4.1 (see section 4.2).

The initial expectation for this experiment was that the modeling would be the most time-consuming task. However, since no test execution environment was in place this was an equally time-consuming task. The modeling of this experiment was the first time the theory from the Qtronic course was applied to this project. Hence completing the model required more time than expected. This was due to the learning phase and the time required for investigating the structures and possibilities within QML. Primarily, modeling the error handling required the most time and included considerations regarding design factors (as discussed in section 4.2.1).

The expectation for the test generation was that full test design configuration coverage would be reached since the model was fairly small and this expectation was fulfilled. With regard to the test generation time and the number of generated test cases there was no expectation. However, the test generation time (13 seconds) and the lines of code for the test suite (2860) illustrate the advantage of MBT, even though approximately one working week in total was required to successfully execute the test cases. Considering one estimated working week, or five working days, to complete this experiment the learning time is estimated as four days. The reason for the estimation of the learning time is that both the main part of the modeling time (two working days) and the main part of the test harness implementation time (two working days) were required for learning.

The test harness and the test execution environment implementation required more thought and reading than actual implementation. The thinking required was primarily to define the test execution environment and how the SUT and the test scripts would communicate. This is indicated by the test harness implementation time (two working days), which included the design of the test execution environment, along with the lines of code for the test harness (99 lines of code) and the test execution environment (73 lines of code). Prior knowledge and experience of socket communication within TCL would have resulted in a significantly shorter period of time for implementing the test harness.

The test scripts generated from Qtronic fulfilled the expectations in terms of the test suite (including the test cases), which was completely rendered and defined by Qtronic. For the test harness implementation it was hard to have any expectations. Nevertheless, the test harness implicitly included oracle information, which was the expected output generated from the model. This was expected as a feature of MBT, even though when using Qtronic it is up to the tester to make use of this information. The oracle information is implicit in the regard that it is known within receiving test harness procedures, which are generated from messages on outgoing (outbound) interfaces as modeled in the QML model. The message structure as such, which is the base of the test harness procedure name, and the message record data members (parameters of the generated test harness procedure) constitute the oracle information. How this information is used is a task of the tester. The reason for this being a task of the tester is flexibility considerations and to support of existing test execution environments within organizations using Qtronic.

The goal of the experiment was achieved since test cases were successfully executed on the SUT while assigning a pass/fail verdict for each test case. To summarize, the initial experiment included a learning phase since this was the first experiment where Qtronic and MBT were applied. This experiment also included defining the test execution environment.

Hence this experiment required more time to complete (approximately one working week), compared to the subsequent experiments. The conclusions of the initial experiment are listed below:

- Design factors are to be considered when creating QML models.
- Modeling the basic system was straightforward while modeling the error handling required more time in terms of evaluating structures and possibilities within QML.
- Most of the test harness implementation time was spent on defining the test execution environment.
- Given a test execution environment the test harness implementation was straightforward, where all procedures either are sending input or retrieving output from the SUT.
- The tester is responsible for implementing oracle functionality in the test harness (i.e. to compare expected and actual SUT output) and thereby also assigning pass/fail verdicts to each test case.
- The initial experiment required approximately one working week and resulted in 25 test cases (defined by 2860 lines of code).
- Most of the time for the initial experiment was learning time (estimated as four working days), primarily required for the creating the QML model and for implementing the test execution environment.
- Encountered test execution failures were either string mismatches (for expected and actual output) or deadlocks (model and SUT not consistent).

The goal of the experiment was reached since generated test cases were successfully executed against the SUT and each test case was assigned a pass/fail verdict.

4.3 Experiment 2

The second experiment included added requirements and an extended specification. The goal of this experiment was to see how the implication of an extended model propagated through the different tasks involved in the Qtronic testing process. The test execution environment was not an issue in this experiment since it was defined and implemented in the initial experiment. The primary focus of the analysis of this experiment was on the model development and the test harness development, and to compare the work of this experiment to the work of the initial experiment.

Measures	Experiment 1	Experiment 2
Modeling time	2 days	1 day
Test generation time	13 seconds	2 min 34 sec
Test design configuration coverage	100%	100%
Number of generated test cases	25	52
Time to implement test harness	2 days	1 hour
Lines of code: Test suite	2860	6278
Number of test harness procedures	18	26
Lines of code: Test harness	99	155
Average: LOC / Harness procedure	5.5	~5.96
Lines of code: Test execution environment	73	73

Table 4.2: Results experiment 1 and experiment 2

The primary focus of the analysis of this experiment is on the modeling and the test harness implementation. Furthermore, the work of this experiment is compared to the work of the initial experiment since the model used in this experiment was based on the one of the initial experiment.

4.3.1 Model Development

The model development of this experiment (described in section 3.5.2) reused the resulting model of the initial experiment. The modeling task included the design of new features, extending the specification (see Appendix C.1.2) and to extend the QML model. Two new features were designed, namely account deposit and account transaction, while considering design factors as discussed in the analysis of the initial experiment. The deposit feature involved the user depositing money into the account associated with the card. The transaction feature involved the transfer of money from the card holder's account to another account (entered by the card holder). The analysis of this experiment primarily focuses on the model development and the test harness development, including comparisons with the initial experiment.

Basic system

The model of this experiment was created using the model of the initial experiment and the second version of the specification (see Appendix C.1.2). The second version of the specification was defined prior to the creation of the QML model. The details of the modeling task are described and discussed in the experiment description (see section 3.5.2). The existing model (from the initial experiment) was structured to encapsulate functionality of the protocol module, namely for the authentication process and the withdrawal process. That is,

functionality as described in the initial specification (see Appendix C.1.1) was encapsulated and described as internal state logic of the top-level state machine (using internal state machines for the Authentication and Withdrawal states).

Using the extended specification (see Appendix C.1.2) the existing QML model was extended to support the new basic functionality (as described in the specification), excluding error handling. The extension included two added states (Deposit and Transfer) to the top-level state machine as well as defining internal state machines associated with these two states. As discussed in the analysis of the initial experiment (see section 4.1), design factors (readability, extendibility and maintainability) were considered when modeling. The modeling practices were outlined in the initial experiment and the QML model of this experiment followed the same practices. Given the experiences and the knowledge gained during the first experiment this extension was completed in a short period of time, now being familiar with Qtronic Modeler and QML.

The modeling of the basic functionality added in this experiment was straight-forward. The design of the model was outlined in the first experiment, thus the extensions modeled in this experiment followed the same structure. The goal of the modeling task was still to create a module structure and to encapsulate behavior of the model to aid readability, extendibility and maintainability. This experiment did not require the same learning phase as the first experiment. Although knowledge about features and structures of QML still required to be learned when modeling the existing model was well-structured and sufficient. The same modeling methodology would have been applied given the new experiences and lessons learned.

The extension of the QML model also included the introduction of data members for message types, such as card number for the message of an inserted card, as described in the experiment description (see section 3.5.2). Adding the data members to the record definitions was completed quickly. However, modeling the dependencies between the parameters of incoming messages and the parameters of the outgoing messages in the graphical notation required more time. For example, the action following the event of entering a withdrawal amount is a network message sent to the server. This network message includes the withdrawal amount as a part of the withdrawal process. This was not a challenging modeling task but it required some time to inspect the model to verify that the dependencies were correctly depicted.

The expectations regarding modeling the basic system of this experiment was that it would require small changes to the existing model. It was expected that the transition definitions needed to be modified to support message parameters as well as extending the message definitions. Furthermore, it was expected that most of the modeling time would be spent on defining the new state machines for account deposits and account transactions.

These expectations were met. The design of the model was outlined in the initial experiment, thus it was a straight-forward task to extend the QML model according to the new requirements. The extension of the top-level state machine required small changes since only two states were added. The creation of the two new state machines was straight-forward since they were constructed using the same structure and methodology as the existing state machines. Creating these two state machines required approximately two hours, including time to inspect and time to test the model. Extending the model to support message parameters required some time, primarily to inspect and verify the dependencies between

messages. This part of the modeling task was completed approximately within one hour (compared to the total modeling time of approximately one working day), since this extension applied to the complete model (i.e. all state machines).

Error handling

The extended model at this point included error handling for the functionality modeled in the initial experiment. Error handling for the new features of the protocol module needed to be added. As discussed in the experiment description (see section 3.5.2), more final states were added for the error handling in this experiment. For each state of the model one final state was added, with a transition from the state to the corresponding new final state. These transitions, for the new final states, handled unexpected events, such as user entered interruptions and network failures. For each state expecting acknowledgements from the server one additional final state was added, with a transition describing the receipt and actions in the event of a negative acknowledgement. This methodology was first applied to the top-level state machine but later also for the state machines bound to states in the top-level state machine to use the same methodology in all state machines. This included modifying the error handling of the authentication state and its internal state machine, which was handled in a uniform way in the initial experiment.

This modeling practice solved the problem of having many transitions ending in the same final state, as discussed in the experiment description (see section 3.5.2). The solution enhanced the readability of the model, both in the top-level state machine and in the state machines used to describe internal state logic, since the basic system and the error handling were separated. The path through the basic system ended in one final state whereas the error handling used separate final states, in contrast to the methodology used in the initial experiment (where all transitions, including the basic system and the error handling, ended in the same final state).

The initial expectation was that the error handling would use the methodology of the initial experiment and not be a time-consuming task. However, issues regarding the readability of the top-level state machine were quickly encountered. The idea of introducing more final states in the model was then adopted and was applied to all state machines. The restructuring of the error handling for all state machines required approximately the same time as the modeling of the basic system. The time to modify the model for this purpose was estimated to about half a working day, or about four hours.

4.3.2 Test generation

The test generation of this experiment was based on requirement statements, as stated in the experiment description (see section 3.5.2). As discussed for the test generation analysis of the initial experiment (see section 4.2.2), the purpose of the project work was not to evaluate test design configuration parameters systematically. Thus requirements were chosen for simplicity. Moreover, since the requirement statements were structured and grouped as described in the experiment description (see section 3.5.2), this made it easier to track the relationship between the generated test cases and the requirements in the model. These relationships were summarized in the traceability matrix (see Appendix B.3 for an example). The requirements were stated in the model based on events (incoming messages to the system) of the model. Thus, all requirements corresponded to system input, as modeled.

Compared to the initial experiment, the requirements were further grouped. At the highest level they were grouped by which interface they originated from (user application interface or network interface). At the next level they were grouped according to which state machine (i.e. authentication, withdrawal, balance, deposit, transaction or top-level) they were used in. This second level of grouping was useful for only generating test cases for a particular functionality of the model, since the test design configuration could be set to cover all requirements, groups of requirements or individual requirements.

The problem of not generating test cases for error events of all states within the authentication state machine, as described in the test generation analysis of the initial experiment (see section 4.2.2), did not apply to this experiment. The error handling of that state machine was restructured to use the same methodology for error handling as the other state machines in this experiment (stated in the model development analysis, see section 4.3.1).

4.3.3 Test harness implementation

In this section the test harness implementation of this experiment is compared to that of the initial experiment. The test harness implementation of this experiment made use of the existing implementation from the initial experiment. An important consideration for the comparison between the test harness of this experiment and the initial experiment was that message parameters were added in this experiment. Consequently such parameters had to be handled in the test harness. For most of the procedures the implementation only required small modifications. This is illustrated in the sample code below.

```
proc userInAmountInput { money } { \
  global sockChan
  set msg "Amount input / $money"
  send $sockChan $msg
}
```

This sample code may be compared to sample code included in the test harness analysis of the initial experiment (see section 4.2.3). The difference in this experiment is that procedure includes the *money* parameter. That was the only modification to the procedure as defined for the initial experiment. The other test harness procedures common for the two experiments were modified in the same way.

The script rendering of this experiment resulted in a test harness file including 8 additional TCL procedures compared to the initial experiment. These 8 procedures were rendered according to new message types defined in the system block of the extended model and all 8 procedures corresponded to modeled messages of the user application interface. The new test harness procedures followed the same structure as illustrated for the modified existing procedures.

However, the procedures corresponding to network messages that carried data required more modification. The sample procedure below is the procedure used for sending network data to the SUT of the initial experiment.

```

proc netInTData { type header } { \
  global sockChan
  set msg "T-Data $type $header"
  send $sockChan $msg
}

```

In the initial experiment message parameters, other than for describing the message type, were omitted. The corresponding procedure of this experiment is illustrated in the sample code below.

```

proc netInTData { type header payload } { \
  global sockChan
  set msg "T-Data $type $header"
  if { $payload != "{}" } {
    set l [split $payload ,]
    for {set i 0} {$i < [llength $l]} {incr i} {
      set msg "$msg / [lindex $l $i]"
    }
  }
  send $sockChan $msg
}

```

Since message parameters were added to the model in this experiment this affected the test harness implementation, which required modifications for this purpose. The expectation was that the existing test harness procedures (from the initial experiment) would not require any modification other than for the added message parameters. Furthermore, a second expectation was that the new procedures of this experiment, corresponding to the new message types of the extended model, would need to be implemented following the same structure as the reused procedures.

The expectations on the test harness of this experiment were fulfilled. The extended model comprised new functionality, including account deposits and account transactions. The model was also extended to include message parameters, such as the entered integer withdrawal amount. Taking only the new functionality of the protocol module into account, and not the introduction of the message parameters, no modifications were required to the test harness of the initial experiment and hence the original test harness could be reused. All test harness procedures implemented in the initial experiment were reused, although a few new procedures corresponding to the new functionality of this experiment required implementation. However, the message parameters added to the model in this experiment required small modifications to use the procedure parameters. The test harness of this experiment required the implementation of the 8 new procedures generated from the new message types, as expected. However that implementation was straightforward since the new procedures followed the same structure as the existing ones.

4.3.4 Test execution

Specific for the test execution of this experiment compared to the initial experiment was that the SUT input and the SUT output, as well as the expected output, included message parameters. Thus a new type of failure occurred, caused by errors in the SUT implementation for parameter dependencies between messages, namely for the parameters of SUT input and the parameters of the following SUT output. Ultimately these failures were detected as string mismatches between actual and expected output in the log file.

As discussed in the in the test execution analysis of the initial experiment (see section 4.2.4), the test execution what was could be expected when performing black-box testing. The generated log file included all input, expected output and actual output of the SUT. This information was sufficient for evaluating the test cases, and the comparison between expected and actual output was performed in the test scripts and mismatches were indicated in the log file. Thus pass/fail verdicts was applied to all test cases in the test suite, indicating which test case to inspect if failures occurred.

4.3.5 Results and conclusions: experiment 2

The results of this experiment and the results of the initial experiment are given in Table 4.2 (see section 4.3).

The expectation for the modeling task of this experiment was that the extended model, according to the added requirements, would use the design and structure of the existing model. Furthermore, the expectation was that the focus of the modeling task would be on creating the two new state machines for the new functionality of the protocol module.

The modeling of the basic system (system behavior excluding error handling) was as expected a straightforward task, given the design outlined in the initial experiment. However, considering design factors when modeling the error handling for the extended basic system issues were encountered, primarily regarding readability (see the experiment description in section 3.5.2 and model development analysis in section 4.3.1). Thus the error handling of the complete model (i.e. all state machines) was restructured to separate the error handling from the basic system. Since the error handling was restructured the modeling task of this experiment was more time-consuming than expected.

The modeling time was estimated to one working day. A significant amount of this time was used for restructuring the error handling of the model, which was not expected before starting this experiment. However, the modeling time of this experiment was still significantly less than for the initial experiment, which required approximately two working days. Hence the expectation that the modeling of this experiment would require less time than the modeling of the first experiment was fulfilled.

The test generation resulted in full test design configuration coverage, as was expected since the model still was fairly small although the number of possible paths through the model was increased. The test generation time differed significantly for this experiment compared to the initial experiment, increasing from 13 seconds to 2 minutes and 34 seconds (see Table 4.2). The test generation time was expected to increase since the model was extended, but there were no expectations regarding as to what extent. The number of test cases increased from 25 of the initial experiment to 52 test cases for this experiment. This was a consequence of the extended model, including the basic system and the error handling for the added functionality.

Since the number of test cases was roughly twice as many for this experiment compared to the initial experiment, the lines of code for the test suite was expected to increase by a similar factor. The test suite file of this experiment included 6278 lines of code, which was relatively close to a factor of two (the test suite of the initial experiment resulted in 2860 lines of code). The lines of code were also affected by the introduced message parameters since they required initializations in each test case of the test suite.

The number of test harness procedures and the lines of code of the test harness are provided as indications of the work effort and implementation required to successfully execute test cases on the SUT. However, most of the procedures implemented in the first experiment were reused. All the 99 lines of code from the initial experiment were reused, although roughly one line of code per reused test harness procedure was slightly modified to support message parameters. Other than the small modifications, the test harness only required implementation for the 8 new procedures corresponding to the extended system block of the model. Moreover, the test execution environment implemented in the initial experiment did not require any further implementation and was completely reused. The total time of the test harness modifications and implementation was estimated to be one hour.

The goal of this experiment was to add requirements and extend to specification compared to the initial experiment to evaluate how the implications of an extended model propagates through the different tasks involved in the Qtronic testing process. This goal was achieved since the implications of the extended model were shown in the test generation, the test harness implementation and ultimately in the test execution. Hence, the relationship between added behavior in the model and required test harness implementation could be evaluated and discussed.

To summarize, the second experiment reused the artifacts of the initial experiment and extended the specification. This experiment still included a learning phase, although not as significant as for the initial experiment. The conclusions of the second experiment are listed below:

- Extending QML model, given design including internal state machines, straightforward since no existing behavior modified.
- Error handling restructured to separate basic system and error handling.
- Modeling required more time than expected to restructure error handling.
- Extended model resulted in a significantly increased number of generated test cases compared to the initial experiment.
- Extended specification and QML model resulted in implementation of 8 new test harness procedures, corresponding to the extended functionality. All procedures from the initial experiment were reused.
- Modified QML model (supporting message parameters) resulted in that roughly one line of code per test harness procedure was slightly modified.
- Test harness implementation required little time (one hour) since most of the test harness implementation of the initial experiment was reused.
- Implementation of the test harness confirmed that the procedures are only dependent on the system block of the model and the defined message types (records).
- Test execution resulted in the same type of failures as the test execution of the first experiment (output mismatches or deadlocks).

The goal of the experiment was reached since the relationship between the extended QML model, the generated test cases and the test harness implementation could be evaluated compared to the corresponding artifacts of the initial experiment.

4.4 Experiment 3

The third experiment included changed requirements and a modified specification (defined in Appendix C.1.3) compared with the specification of the second experiment (defined in Appendix C.1.3). The goal of this experiment was to introduce new requirements for the authentication and to modify the specification to see how the implication of modifications in the existing model propagated through the different tasks in the Qtronic testing process. The new requirements were that a biometric authentication was obligatory for carrying out account requests (account withdrawal, account balance, account deposit and account transaction). These new requirements also resulted in the goal to create a reusable QML structure that could be applied to all these different account request processes. The focus of the analysis of this experiment is on the model development and the test harness implementation, and to compare the work of this experiment to the work of the second experiment. Furthermore, the analysis also focuses on the goal of creating a reusable QML structure that could be applied and used in several state machines.

Measures	Experiment 2	Experiment 3
Modeling time	1 day	4 hours
Test generation time	2 min 34 sec	3 min 11 sec
Test design configuration coverage	100%	100%
Number of generated test cases	52	56
Time to implement test harness	1 hour	10 min
Lines of code: Test suite	6278	7540
Number of test harness procedures	26	28
Lines of code: Test harness	155	165
Average: LOC / Harness procedure	~5.96	~5.89
Lines of code: Test execution environment	73	73

Table 4.3: Results experiment 2 and experiment 3

The primary focus of the analysis of this experiment is on the model development and the test harness development, and to compare the work of this experiment to the work of the second experiment.

4.4.1 Model Development

The QML model of this experiment was created using the QML model of the second experiment and the modified specification (defined in Appendix C.1.3). The new requirements for authentication did not apply to the general authentication process already modeled (using an internal state machine). The new requirement was a biometric authentication for completing account requests and was defined to be performed before an account request was carried out by the server. For example, the biometric authentication was defined to be performed after entering a withdrawal amount but before the withdrawal was performed. Thus the biometric authentication needed to be modeled inside the state machines describing account request logic (i.e. internal state machines for withdrawal, balance, deposit and transfer). Hence the goal was to apply a reusable QML structure, i.e. to define the biometric authentication state machine and to reuse it within the four state machines instead of repeating the same logic for all state machines.

Basic system

The primary goal of the modeling task was to apply a reusable QML structure so that modeled behavior could be used in several QML state machines. The solution was to create a new and separate QML model, including a state machine for the biometric authentication logic. This model was a simple state machine including only three states (defined according to the specification in Appendix C.1.3). The biometric authentication state machine was then used in the state machines for withdrawal, balance, deposit and transaction requests, as illustrated in the experiment description (see section 3.5.3). The balance functionality of the protocol module, only including one state, was in this experiment modeled as an internal state machine to apply the same methodology for all account requests.

The use of the biometric authentication state machine was synchronized so that it had to be completed for the state machines to continue their execution (see details of the synchronization in section 3.5.3). Through this solution the top-level state machine required no modification. The internal state machines only required one added QML code block as an entry action of one state for each state machine. Thus the existing logic within the internal state machines was not altered.

The solution was a good way of using general logic in different state machines instead of repeating this logic, only requiring simple synchronization in the existing state machines. Before starting to extend the QML model for this experiment the expectation was that QML would support the reuse of general logic, although not knowing how to achieve this. Most of the modeling time of this experiment was spent on reading the Qtronic User Manual to find a solution. However, the solution was outlined through email conversation with the Conformiq instructor that helped creating the initial QML model. Once knowing that a separate QML model could be used in the existing QML model the modeling task was fairly straightforward and not time-consuming. Most of the modeling time was spent on synchronizing the use of the new QML model (including the biometric authentication state machine).

The structuring of biometric authentication process was a result of considering design factors, primarily extendibility. The new QML model, including a state machine, could be reused in any state machine thus illustrating an advantage in terms of extendibility. Moreover, the solution was also an advantaged in terms of readability and maintainability since the abstraction level of the new requirements in the model was increased and would only required modification of that particular state machine. Thus the expectation that QML would support a structure for the reuse of behavior was fulfilled.

Error handling

As stated for the basic system, the existing logic in the QML model, as defined in the second experiment (see experiment description in section 3.5.2 and model development analysis in section 4.3.1), required no modification. This applied to the error handling as well. However, error handling for the acknowledgement of the biometric authentication was required for each state machine using the biometric authentication state machine. This error handling was added for the state including the code block starting the biometric authentication state machine. Hence, a total of four error handling transitions were added to the model compared to the second experiment. The only other error handling modeled for this experiment was within the new biometric authentication state machine, using the same methodology as described for the second experiment.

The expectation for the error handling, given the solution using a separate QML model and only considering the QML model of the second experiment, was that only error handling for the synchronization of the new state machine was required. This expectation was fulfilled and the modeling of the error handling of this experiment was straightforward given the structure outlined for the existing state machines.

4.4.2 Test generation

The test generation of this experiment was, as for the previous experiments (see sections 3.5.1 and 3.5.2), based on requirements stated in the model. As discussed in the test generation of the experiment description and as in the test generation analysis of the second experiment, requirements were grouped by interface at the highest level and by state machine at the second level. A new group of requirements were added at the second level in this experiment, namely for the biometric authentication state machine, and these requirements were stated in that state machine.

The test generation of this experiment resulted in 56 generated tests. That was only 4 additional test cases compared the second experiment. The reason for this was that the biometric authentication process was obligatory for all account requests. Thus only adding the basic system for this state machine would have resulted in the same number of generated test cases for this experiment as for the second experiment. The reason for the 4 additional test cases was that the biometric authentication process only could fail in particular scenario (mismatching biometric information in server comparison). Since the biometric authentication process was applied for the four possible account requests (i.e. internal state machines) this error could occur for each of these four state machines, thus resulting in four additional test cases.

However, the amount of interaction (i.e. number of incoming and outgoing messages) increased for each test case including the biometric authentication. The additional sequences of messages were a result of the incoming and outgoing messages of the biometric authentication state machine.

4.4.3 Test harness implementation

The fact that the QML model of this experiment used a separate QML model for the biometric authentication process was not visible at the test harness level, as expected. The script rendering resulted in a test harness that included two new procedures compared to the test harness of the second experiment (compare experiment results in Table 4.3). These two procedures were rendered as a result of the extended system block in the model of this experiment. The procedures corresponded to messages for the user application interface used in the biometric authentication state machine.

Since this experiment only modified the internal state machines to use the QML model for the biometric authentication and to handle the negative acknowledgement of the biometric authentication, the existing test harness implementation from the second experiment (see section 3.5.2) was reused completely. The only test harness implementation required in this experiment was for the two new procedures, which followed the same structure as the existing procedures. Thus the test harness implementation task of this experiment was trivial task and was completed in 10 minutes.

4.4.4 Test execution

The test execution of this experiment did not result in any new experiences and conclusions compared to the test execution task of the second experiment. The difference was that the test cases of the test suite in general included more interactions between the SUT and the test scripts. This was a consequence of the biometric authentication added to the model in this experiment.

The test execution resulted in pass/fail verdicts for each test case. The generated test execution log file, as in the second experiment, included message parameters for the SUT input, the SUT output and the expected output. The initial test execution of this experiment resulted in failures for all test cases including the biometric authentication. This was due to that the use of the biometric authentication (included in the new QML model) was not synchronized for the state machines defining account requests (i.e. withdrawal, balance, deposit and transfer). Hence, the model was inspected and the error was found. The error in the model was a misuse of a Boolean data member used for the synchronization. The error was corrected and no further failures were detected during test execution.

4.4.5 Results and conclusions: experiment 3

The results of this experiment are given in Table 4.3 (see section 4.4).

The expectation of the modeling task in this experiment was that QML would support a structure for reusing general logic in several state machines. When starting the modeling task of this experiment, the Qtronic User Manual was read to find a suitable solution for this purpose. Hence, there was initially no expectation regarding the modeling time. However, when the solution was found, through email conversation with the Conformiq instructor, the expectation was that the modeling of this structure and the small modifications to the existing model would be completed in a short period. The solution only regarded one simple QML model (including a state machine) and the use of this new QML model in the existing model.

The modeling time of this experiment was estimated to a total of 4 hours, including the time reading the Qtronic User Manual in the search for a solution. The modeling time of this experiment was hence significantly less than the modeling task for the second experiment, which required approximately one working day. The resulting model of this experiment was a good solution considering design factors. By using a separate QML model for the added general functionality the abstraction level of the complete model was increased. The readability model was intact since the existing state machines only required small and specific modifications in order to use the new QML model. The solution was also good practice for extendibility and maintainability since behavior could be encapsulated in separate QML models and reused in several state machines.

As for the previous experiments the test generation resulted in full test design configuration coverage. This was expected since the model still was fairly small. The test generation time was increased since the model was extended. The test generation of this experiment was completed in 3 minutes and 11 seconds, compared to 2 minutes and 34 seconds for the second experiment. The number of test cases increased from 52 for the second experiment to 56 of this experiment. These 4 additional test cases included the error handling for the use of biometric authentication state machine. The other 52 test cases were similar to the test cases generated in the second experiment, with the difference that most of the test cases of this experiment included biometric authentication.

The rendered test suite file included 7540 lines of code, compared to 6278 for the second experiment. The reason for the increased number of lines of code was that the test cases of this test generation generally included more interaction (i.e. SUT input and expected output) as a result of the biometric authentication.

The number of test harness procedures and the lines of code for the test harness are provided as indications of the work effort required to successfully execute test cases on the SUT. The test harness of the second experiment was reused without modifications in this experiment. Only two test harness procedures required implementation (as discussed in section 4.4.3). This is illustrated by the test harness implementation time, which was estimated to 10 minutes for this experiment. The lines of code for the test suite (7540) considering the modeling time (estimated to 4 hours) and the test harness implementation time (estimated to 10 minutes) illustrates the gain of MBT.

The goal of this experiment was to change the authentication requirements and to modify the specification to evaluate how the implication of how modified requirements in the model propagated through the task involved in the Qtronic testing process. Furthermore, the goal of this experiment was to model a structure that could be reused in any number of state machines. Both these goals were achieved, although related, since the changed requirements of this experiment were encapsulated in a reusable structure.

To summarize, the third experiment reused the artifacts of the second experiment and modified the specification. The modification of the specification included a required biometric authentication for all account requests. The conclusions of the third experiment are listed below:

- Discovered a reusable structure within QML where a separate QML model was used within the existing QML model, thus encapsulating logic which could be reused in any number of state machines.
- Result of the solution using a separate QML model was that the existing model only required little new logic to support the new general logic.
- Most of the modeling was spent on finding a solution within QML to reuse logic in several state machines.
- The modeling solution was good considering design factors (extendibility and maintainability), where logic is encapsulated in different QML models and hence may be tested separately. Thus also supported reusability.
- Modeling solution resulted in increased level of abstraction.
- Small increase for number of test cases and test generation time.
- Complete test harness of the second experiment reused.
- Test harness implementation only included two new procedures corresponding to messages within the biometric authentication process.
- Test harness only dependent on system block and message types, hence not affected by changes of the behavior (i.e. transitions).
- Test execution resulted in the same type of failures as the first two experiments, namely output mismatches and deadlocks.

The goal of the experiment was reached since the relationship between the modified QML model, the generated test cases and the test harness implementation could be evaluated compared to the corresponding artifacts of the second experiment.

4.5 Experiment 4

The fourth and final experiment included the new requirements (biometric authentication) and was based on the same specification (see Appendix C.1.3) as defined in the third experiment (see section 3.5.3). The goal of this experiment was to evaluate the implications of moving logic from the model to the test harness, while still testing the same version of the SUT as in the third experiment.

Measures	Experiment 3	Experiment 4
Modeling time	4 hours	2 hours
Test generation time	3 min 11 sec	3 min 6 sec
Test design configuration coverage	100%	100%
Number of generated test cases	56	56
Time to implement test harness	10 min	2 hours
Lines of code: Test suite	7540	7476
Number of test harness procedures	28	28
Lines of code: Test harness	165	229
Average: LOC / Harness procedure	~5.89	~8.18
Lines of code: Test execution environment	73	73

Table 4.4: Results experiment 3 and experiment 4

The primary focus of the analysis of this experiment is on the abstraction level of the model and consequently on the test harness implementation. Furthermore, the analysis will compare the work of this experiment to the work of the third experiment since they were based on the same specification and used the same version of the SUT.

4.5.1 Model Development

The QML model of this experiment was created using the QML model of the third experiment. Since the goal was to evaluate the implications of moving logic from the model to the test harness the model development of this experiment included removing logic from the model of the third experiment.

The testing goal of the experiment was still to achieve the same amount of testing as in the third experiment, since the same version of the SUT was used, but in a different way. An important consideration for this experiment was also that the test generation should include the same message sequences (of incoming and outgoing messages) and the corresponding parameter values as generated by Qtronic for the third experiment. Hence no significant modifications to the structure of the model could be made, such as modifying event and action messages or modifying message types. Thus a part of the experiment was to evaluate what logic of the existing model that could be removed and implemented in the test harness.

The only logic that could be removed from the model was the dependencies between message parameters. That is, in the model of the third experiment parameters of an outgoing message (action) are in a number of cases dependent on parameters of the incoming message (event). For example, given an incoming message to the protocol module from the user application interface indicating an entered withdrawal amount the protocol module sends a network message to the server. The parameter of the network message sent to the server is the

payload, which in this particular scenario includes the withdrawal amount as provided by the parameter for the event. These modeled dependencies between events and actions were removed from the action part of transitions in the model.

Furthermore, the model of the second and third experiment included code blocks (require statements) that were used to describe the dependencies between parameters of different incoming messages (see section 3.5.4). An example is the dependency between an incoming message on the user application interface indicating an entered withdrawal amount and an incoming message on the network interface indicating the acknowledgement of a withdrawal. In this case a transition guard was used for the incoming network message to describe that the acknowledged withdrawal amount had to be the same as the withdrawal amount entered by the user. Require statements describing such dependencies were also removed from the model.

To summarize, the basic system and the error handling as defined in the third experiment were intact and not modified in this experiment. The logic removed from the model described dependencies between parameters of an incoming message and the parameters of the following outgoing message as well as dependencies between parameters of different incoming messages.

The expectation regarding the modeling time required to remove the modeled dependencies was that this would be an easy task to complete and require little time. This was true but the modeling task required time to inspect the new model for verifying that all modeled dependencies were removed and to verify that no other logic was removed.

4.5.2 Test generation

The model modified in this experiment was very similar to the model of the third experiment since only specific action logic for a number of transitions were removed. Hence the test generation was very similar to the test generation of the third experiment (see section 4.4.2).

The test generation was based on requirements stated in the model, as for the previous experiments. The requirements of this model were exactly the same as for the model of the third experiment. Thus experiments were grouped by interface at the highest level and by state machine at the second level.

The test generation of this experiment resulted in 56 generated test cases, as expected. The number of test cases was expected since the same amount of test cases was generated in the third experiment. The modifications of the model in this experiment did not include any modifications for the requirement statements. Hence the same amount of test cases was expected.

4.5.3 Test harness implementation

The test harness of the third experiment was reused in the test harness implementation of this experiment. Since logic describing dependencies between message parameters was removed from the model this logic had to be implemented in the test harness.

As illustrated in the experiment description (see section 3.5.4) this logic was implemented by using global variables in the TCL implementation of the test harness. These global variables were always set in test harness procedures sending input to the SUT (corresponding

to incoming message in the model). The reason for this was that the model only included transition guards (see Qtronic Modeler in section 3.2.1) for incoming messages to control parameter values in the test generation. Most of the test harness procedures receiving SUT output and defining expected SUT output (corresponding to outgoing messages in the model) used global variables in their definitions. The reason for this was that the parameters of the outgoing messages often were dependent on the parameters of the incoming messages, as discussed in the model development analysis of this experiment. However, one test harness procedure sending input to the SUT also used the global variables in its definition. This was the procedure for sending network data to the SUT (illustrated in the test harness sample code in the experiment description). The reason for this procedure using a number of global variables was to implement the logic for dependencies between parameters of different incoming messages to the SUT, as discussed in the model development analysis.

The expectation for the test harness implementation of this experiment was that it would require some implementation for describing the dependencies between message parameters. Since these dependencies were implemented using global variables the task was expected to require a significant amount of time. However, the test harness implementation was completed within two hours, including time to correct errors discovered during test execution. The completed test harness included 229 lines of code, compared to the 165 lines of code reused from the third experiment. Some procedures required almost no modifications whereas others were required to be completely re-implemented. An example of the latter is the test harness sample code in the experiment description (see section 3.5.4).

4.5.4 Test execution

As for the third experiment, the test execution of this experiment did not result in any new experiences and conclusions compared to the test execution task of the second experiment.

Compared to the third experiment, the test execution task of this experiment was very similar since the same version of the SUT was tested. The difference was that failures due to errors in the dependencies between parameters of different messages were more difficult to detect by inspection. The reason for this being difficult to detect by inspection was that those dependencies were defined in the test harness in this experiment, and not in the model as in the third experiment. The log file was particularly useful for the analysis of this experiment since the errors could be traced back to the first occurrence of particular parameter value.

4.5.5 Results and conclusions: experiment 4

The results of this experiment are given in Table 4.4 (see section 4.5).

The model development of this experiment used the model of the third experiment. The goal was to remove logic from the model and to implement this logic in the test harness, while still testing the same SUT version as in the third experiment. The conclusion from analyzing the model of the third experiment was that the only logic that could be removed was the dependencies between parameters of different messages in model (see section 4.5.1 for a more detailed discussion regarding the removal of logic).

The expectation on the modeling task in this experiment was that it would be an easy task to remove the modeled dependencies between parameters of different messages. Thus the modeling task was expected to be completed within an hour. However, time was required to inspect the new model to verify that all modeled dependencies were removed and that no

transition guards used to manipulate the parameter values in the test generation was removed. The modeling time, including this inspection, was estimated to a total of two hours.

The test generation resulted in full test design configuration coverage and the generated test cases were almost identical to the test cases of the third experiment. The difference was that the test cases of the third experiment included and described the parameter dependencies between different messages. The test generation time of this experiment was 3 minutes and 7 seconds, compared to 3 minutes and 11 seconds for the third experiment.

The rendered test suite file included 7476 lines of code, compared to 7540 for the third experiment. There was no significant difference in the lines of code for these two experiments, as expected. The few additional lines of code in the test suite of the third experiment defined the parameter dependencies. The difference of 64 lines of code spread across 56 test cases.

The test harness implementation time was expected in the region between two and four hours. This task was estimated to have been completed in two hours. The numbers of test harness procedures are the same for this experiment as for the third experiment since they both model the same system behavior. However, since parameter dependencies were implemented in the test harness of this experiment the lines of code for the test harness increased significantly compared to the third experiment. The test harness of this experiment included 229 lines of code, whereas the test harness of the third experiment included 165 lines of code. The test execution resulted in a number of failures due to errors in implementation for the parameter dependencies, as expected. However, using the log file these errors were corrected.

The conclusion of this experiment is that logic may be removed from the model and implemented in the test harness, while still successfully testing the SUT. Hence the goal of this experiment was achieved. However, implementing system behavior in the logic is not a good practice. The model used for MBT should, if possible, depict the complete system behavior required to test the SUT. This only illustrated that it is possible to successfully test the SUT by implementing system behavior in the test harness.

To summarize, the fourth and final experiment reused the artifacts of the third experiment, including the specification and the SUT. Logic from the QML model of the third experiment was removed and instead implemented in the test harness. The conclusions of the fourth experiment are listed below:

- Only logic describing dependencies between parameters of different messages could be removed from the QML model, given that the same features of Qtronic would be used and that this experiment would test the same SUT version as in the third experiment.
- The same number of test cases was generated as for the third experiment, with the difference that the test cases did not include mentioned dependencies.
- There was no significant difference between the generated test suites compared to the test suite of the third experiment.
- The test harness primarily required more implementation (compared to the test harness of the third experiment) for the procedures corresponding to the generic network messages.

- The test harness required global data members to implement the parameter dependencies.
- The test execution of this experiment resulted in significantly more failures that were due to errors in the test harness, compared to the previous experiments.
- The errors resulting in test execution failures were more difficult to detect in this experiment since the errors occurred in the test harness implementation.
- The goal of this experiment was reached since it was shown that logic removed from the QML model could be implemented in the test harness while still successfully testing the same version of the SUT (as in the third experiment).
- Implementing logic in the test harness deviates from the purpose of MBT, which aims at including all logic in the model and to generate test cases from that model.

4.6 Summary of Experiment Results

This project comprised four experiments. The first, the second and the third experiment involved incremental development of the SUT and modifications to the specification. The initial experiment was based on the initial specification, which included withdrawal and balance requests. In the second experiment the specification was extended to include deposit requests and transaction requests. The third experiment included a modified specification to require a successful biometric authentication to complete all account requests. The fourth experiment evaluated the possibility of moving logic from the QML model to the test harness implementation, while using the same version of the specification and the same version of the SUT as for the third experiment. The results of the experiments are given in Table 4.5 below.

Measures	Exp 1	Exp 2	Exp 3	Exp 4
Modeling time	2 days	1 day	4 hours	2 hours
Test generation time	13 seconds	2 min 34 s	3 min 11 s	3 min 6 s
Test design configuration coverage	100%	100%	100%	100%
Number of generated test cases	25	52	56	56
Time to implement test harness	2 days	1 hour	10 min	2 hours
LOC: Test suite	2860	6278	7540	7476
Number of test harness procedures	18	26	28	28
LOC: Test harness	99	155	165	229
Average: LOC / Harness procedure	5.5	~5.96	~5.89	~8.18
LOC: Test execution environment	73	73	73	73

Table 4.5: Results summary

Table 4.5 includes the results of the four experiments. Test generation time, test design configuration coverage and the number of generated test cases are derived from Qtronic. Modeling time and test harness implementation time were estimated during the work. Measures of the test scripts were taken after completing the experiments.

4.7 Project Analysis

In this section the general project work will be analyzed as well as the MBT tool Qtronic and its separate modeling tool, Qtronic Modeler.

4.7.1 Project work

The general goal of this project was to evaluate the concept of MBT. This goal included a theoretical part, including a pre-study, and an experimental part. The experimental part of the project was a feasibility study of MBT. The goal of the feasibility study was to apply the concept of MBT by using a specific tool (Qtronic) to successfully execute tests on an existing test object.

The pre-study of the project initially included a study of the MBT concept, which was an important task to understand its background and its scope. Furthermore, it was important to review reported results and findings from the testing community since MBT is a new concept within software testing. The pre-study resulted in basic knowledge about the process, the benefits and the limitations of the MBT approach. A second part of the pre-study included the goal to learn a MBT tool to be used in the feasibility study, namely Qtronic. Before having access to Qtronic its user manual [8] was used to gain basic knowledge about the features and settings. The learning phase of applying Qtronic started with an introductory course, held by instructors from Conformiq, the company developing the tool. When the course was finished two goals of the project were fulfilled, namely to learn the concept of MBT and to learn to use a MBT tool. After the course the feasibility study of the project begun.

The feasibility study of the project included four experiments. Furthermore, the study included a number of general goals, which applied to each of the experiments. The first goal was to implement a test object according to the specification. The next goal, after implementing the test object, was to develop a model of the test object according to the specification and subsequently generate test cases from that model. The generated test cases were then rendered to an executable format. The next goal of the process was to develop the test execution environment between the test scripts and the test object in order to execute the test cases. The implementation of the test execution environment was an initial effort and the implementation was reused throughout the project. Another goal was to implement the glue code, or the test harness. The test harness implementation used the defined test execution environment and this implementation task was repeated for each version of the model. The last two goals of the process were to execute the tests and to evaluate the results.

The goals of the feasibility study were repeated for each experiment, since the study included incremental development. All four experiments were successfully completed, i.e. all versions of the test object were successfully tested. Thus the general goals of the feasibility study of the project were fulfilled since all sub-goals were necessary to successfully execute the generated test cases. Although the complexity of the test object was fairly small the incremental development throughout the experiments resulted in different experiences and findings for each experiment. By incrementally developing the test object the relationship between the model and the test harness could be investigated, which resulted in a better understanding of how the different tasks in the Qtronic testing process were related.

The main problem of the feasibility study was that the Qtronic course was held fairly late in the project. However, the Qtronic license was received a week in advance so there was some time to get familiar with the tool prior to the course. The fact that the course was held fairly late in the project resulted in a limited time for the experiments of the feasibility study. Thus the feasibility study did not cover all planned goals regarding Qtronic, such as to evaluate all different types of test generation criteria in the tool. Instead one particular test generation criteria was used in all four experiments. An early problem in the feasibility study was that since the project included development of the test object, no test execution environment was in place when the first model was completed. This required some time to define and would not have been a problem if one had existed at the start of the project. The particular problem regarded synchronization of the communication between the test scripts and the test object when executing test cases. However, when the problem was solved the solution was reused throughout the project and did not cause any further problems. The largest problem throughout the project work was the learning phase of the modeling task. Throughout the project work new modeling features were encountered. Thus it was a tradeoff between reconstructing the model, which implicated more of the already limited time, or to proceed with the existing model although better solutions were discovered. Due to the limited time frame the model was only reconstructed when necessary. However, newly discovered modeling features were applied if the model was extended. The modeling problem of the feasibility study was hence primarily due to the learning phase, as may have been expected. Therefore an important conclusion of this project is that modeling experience is crucial when adopting MBT in projects involving complex systems. In this project, the modeling task was the most challenging task whereas the test harness implementation was very similar for each experiment and followed the same pattern.

As mentioned, the modeling task is the most crucial part in the MBT process. If adopting MBT in the organization at Tieto, experience will be important. Furthermore, the design of the model will be important. Since the model is the most important artifact of the MBT approach the quality of the model will be important, especially for larger and more complex systems. The model increases the abstraction level of the testing process and may thus be used as a more effective means of communications between testers, compared to the now used, and manually written, test scripts. The model may be reused for any number of test generations and may possibly be valid for a long time. Moreover, a model of a complex system may involve a large number of testers, or even system developers. Thus a good design of the model is important. The design of the model may also affect test generation times for complex systems, which may be an important factor. However, this thesis has shown that parts of a system may be modeled and consequently tested. Hence, if systems tend to be too large and complex to model parts of the system may be modeled and tested.

A test object of small complexity was a benefit in some aspects. The relationship between the different tasks in the Qtronic testing process was easier to track than what would have been the case for a significantly more complex system. Hence a change in the model, which propagated to the test harness level, was easier to track and thus aided the understanding of the Qtronic testing process. Furthermore, if a significantly more complex system had been used the modeling time would have increased significantly. The goal and purpose of this project was a feasibility study, i.e. to prove the concept and to document experiences. A more complex system would have required a lot more time to create a working model and less time to experiment with the model and the test object. Much of the effort in that case would have been required only to successfully test the test object and it would have been hard to find time to manipulate the test object and the model due to the limited time frame. A project including

a more complex system would have produced results more applicable to projects within Tieto's organization, in terms of modeling time, test generation time and test harness implementation time. However, in such a project much of the time would have been spent on the initial model since the project included a learning phase. Nevertheless, the experiments of this project gave indications of what tasks that are time consuming and indications of the gain when adopting MBT.

The scalability of this project is difficult to approximate. Regarding the two perhaps most important results, the modeling time and the test generation time, the results of this project are not applicable to projects at Tieto. Such results and indications must come from the experience of adopting MBT in the projects. Adopting MBT in such projects would also include a learning phase, thus the complete potential gain is unlikely to be seen immediately. The experience of this project showed that a complete system may be modeled and tested, but also that parts of a system may be modeled, depending on the testing goals. A significantly more complex system, compared to the test object in this project, may result in significantly larger test generation times. One of Tieto's customers has reported test generation times of approximately two days. Such test generation times may be inevitable, but an experience of this project is that parts of the system may be modeled. Thus the model would be simplified and ultimately result in smaller test generation times than modeling the complete system.

The experiences and results of this project indicate that MBT is worth the effort. Although a relatively small test object was used the results of each experiment illustrate the gain of MBT. The MBT process as described in this thesis includes some implementation, but in comparison to manually written test scripts little implementation effort is required. The MBT approach to testing increases the abstraction level of the tester, thus requiring different skills compared to a scripting-based testing approach. The focus was on the modeling task rather than the implementation task, even though some implementation was required. The test harness implementation task of the Qtronic testing process was in this thesis straight-forward when the test execution environment was set up. The modeling task included a learning phase in all experiments of this project, thus more time were required, but still produced a large number of test cases. Moreover, a working and correct model does not require change and may be used in a number of test generations. By simply changing the test generation criteria all kinds of test cases may be generated and the result is sets of test cases, generated dependent on the testing goals.

4.7.2 Qtronic

Qtronic was the MBT tool used in this thesis. Before the Qtronic course was attended (as stated in section 3.2.1) the tool had been installed and examples shipped with the tool had been reviewed. Thus the main features of the tool were known prior to the course. Testing the tool prior to attending the course was useful since the examples resulted in questions and basic knowledge of the tool features. Given the basic knowledge it was easier to focus on details and specific questions during the course.

As expected, the main task of using Qtronic is the modeling. For this purpose QML is used, which is an object-oriented language based on Java, although the standard library of QML is very limited compared to the standard library of Java. The QML models may be expressed entirely in textual notation or together with graphical notation. The graphical notation of QML models is defined using a separate tool, Qtronic Modeler (see section 4.7.3 for analysis).

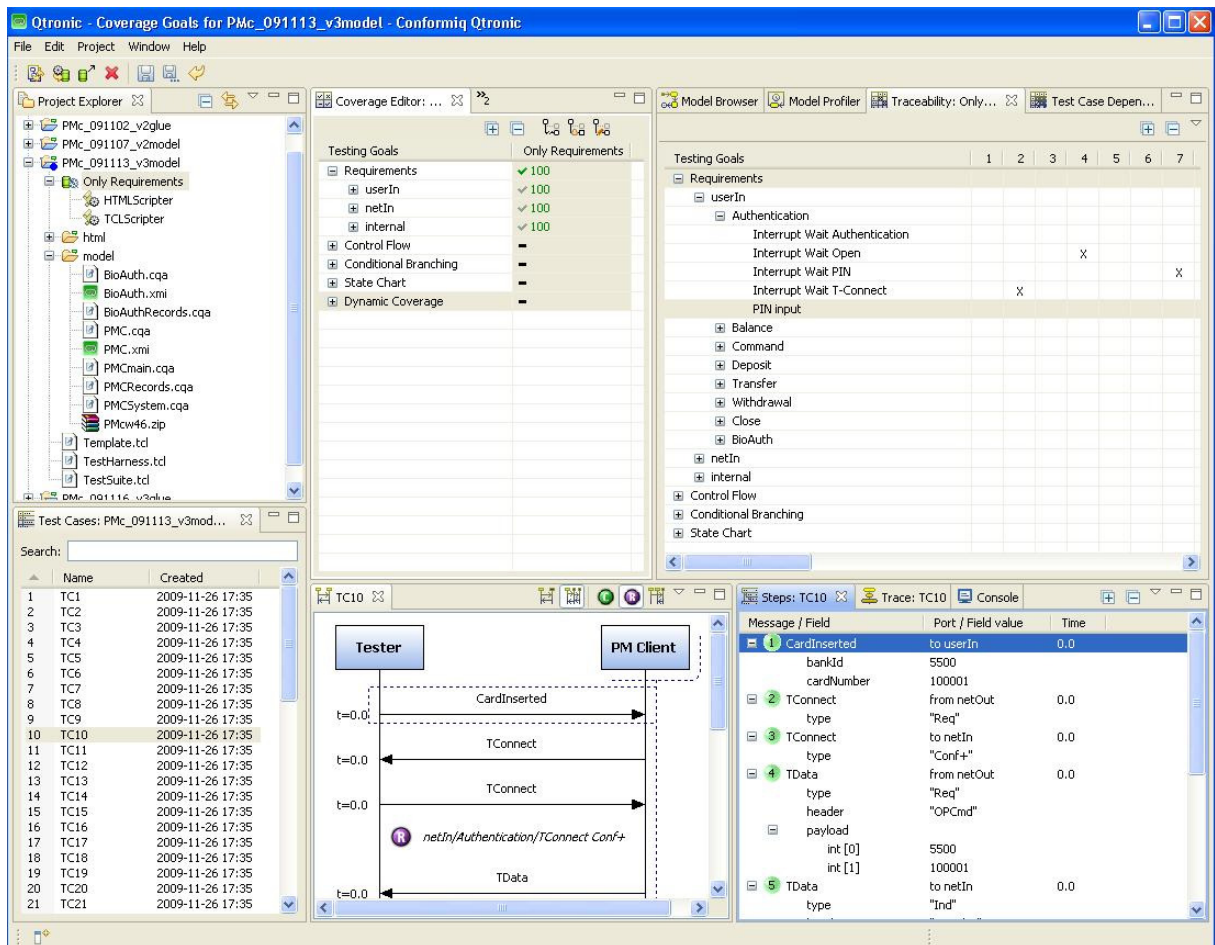


Figure 4.1: Qtronic2 Client

Qtronic Projects

The first step when testing with Qtronic was to create a project. A new project included a test design configuration and a model folder. Graphical and textual QML files were then added to that folder or to some created folder. Hence one or more folders were used to structure the model files of a project. Moreover, model files may be linked between different folders if necessary, to support reusability. For example, if the model is a client-server architecture, two separate folders may be created for the client and for the server. A general folder containing links to the client and the server files, along with a system definition, may then be used define how the two interact. Hence, the conclusion is that Qtronic supports a good way of organizing and structuring models.

Test Design Configurations

Test design configurations (see Appendix C.2.1 for parameters) may include one or more scripter plugins for rendering the generated abstract test cases to an executable format. Each Qtronic project had at least one test design configuration, to which the generated test cases were bound. The test design configuration parameters supported test generation criteria such as different types of state chart coverage, control flow, boundary value analysis and equivalence class partitioning (see Appendix C.2.1 for complete list and details). In this project another test design configuration parameter was used for test generation, namely requirement statements. They are explicitly stated and named in either in the graphical or

textual model files. The goal was to cover all paths of the model and this goal could have been achieved by using state chart coverage parameters. However, requirement statements were chosen because of traceability. The requirement statements were grouped by interface of the model and by state machine and were stated for each transition on the form: *<interface>/<state machine>/<event>*. Thus they provided a mechanism to control test generation, where the test generation could focus on particular state machines, as well as a mechanism to trace the relationship between test cases and covered events in the model. As expected, Qtronic provided a wide range of test configuration parameters, although not all of them were used and explored in this project. The conclusion is that Qtronic supports the test generation criteria that could be expected, although not all parameters were evaluated in this project.

Test Generation Options

The third information of Qtronic projects, other than test design configurations and model files, were the test generation options. In contrast to the test design configuration these options are global for the project and across different test design configurations. An example of such a project option is the lookahead depth, or the search depth (see Appendix C.2.2 for details of these options). The algorithmic options, global for each Qtronic project, defined heuristics used in the test generation, i.e. how test cases are generated. Another useful test generation option was the only finalized runs option. This setting ensured that only test cases ending in a final state, i.e. complete paths through the model were generated. The conclusion regarding the test generation options is that they provide a good mechanism for controlling the test generation, depending on the structure and the complexity of the model.

Test Generation

Qtronic uses a client-server architecture, where the client user interface is an Eclipse-plugin or a stand-alone version. The server component, Qtronic Computation Server, may be run locally or remotely. The computation server is used for test generation and performs the calculations. In this thesis the computation server was set up locally. The test generation started by loading the model files to the computation server and once a test design configuration was defined the test generation was performed. The expectation was that the test generation status would be shown, since test generations may require a significant amount of time. This expectation was fulfilled since the console window reports the current coverage (according to the specified test design configuration) and also reports the approximated time left. However, the reported estimations of the test generation time was rarely accurate. Qtronic also provided a view, the Model Profiler, to track the test generation. This view included information about the test generation, such as which part of the model required the most time. This information was updated continuously during test generation and in the cases where the model contained errors, which were not discovered during the model parsing, this information was helpful discovering such errors. The conclusion of the test generation is that Qtronic provides sufficient information for the test generation although the estimated time left should not be trusted. Furthermore the Model Profiler view is helpful to discover errors in the model that result in unexpectedly large test generation times.

Script Generation

As known prior to the Qtronic course, the tool generated abstract test cases. These abstract test cases were then rendered to an executable format, which in this project were TCL scripts. Qtronic includes scripter plugins for TCL, TTCN-3 and Perl but also provides the possibility

of defining own scripser plugins. The TCL scripser plugin generated a test suite file, including the test cases, as well as a test harness. The test harness included empty procedures, which required implementation. The procedures of the test harness were used to define the test cases of the test suite. Each test harness procedure corresponded to a message type on a particular interface, as defined in the QML model. Thus the procedures were either sending input to or receiving output from the SUT, and the implementation made use of the defined test execution environment. The receiving procedures of the test harness constituted the oracle information, i.e. contained the expected output of the SUT. Thus, when receiving the SUT output it could be compared against the expected output and indicating mismatches. In this way each test case was assigned a pass/fail verdict, although the verdict functionality had to be implemented manually. The expectation of the script generation was that the test suite script would be affected by changing the flow of the model whereas the test harness script would only be affected by changes in the system block, i.e. defined message types of each interface. This expectation was fulfilled, thus most of the test harness was reused throughout the project. The conclusion of the script generation is that the test harness implementation is a straightforward task, given a test execution environment, and provides sufficient means to implement verdict functionality, i.e. oracle information, and that the test suite script is completely generated by Qtronic (the test suite script implementation is based on the test harness procedures).

Qtronic Views

Qtronic included a number of views. Except for the coverage editor (defining the test design configuration settings) and the console, where the progress of the test generation was shown, most views were related to inspecting the generated test cases. A traceability matrix view illustrated the relationship between coverage settings (i.e. test design configuration parameters) and the individual test cases. This view was very useful when the number of generated test cases increased and particular test cases were inspected. The generated test cases, bound to a particular test design configuration, were listed in a separate view. In this view a particular test case could be chosen for inspection. The test case view illustrated the interaction between the tester (Qtronic or the test scripts) and the SUT for a given test case. The view displayed the interactions as input and expected output from the SUT in terms of a sequence diagram. A similar view, the test step view, displayed further details about a particular test case. In this view message parameters could be inspected. Moreover, Qtronic provided an execution trace view that links a particular test case back to model from which it was generated (as a sequence of states in the state machine). This link is also illustrated in a graphical representation of the model. The conclusion regarding the Qtronic views is that they collectively provide a sufficient support for test generation result analysis.

4.7.3 Qtronic Modeler

Qtronic Modeler is a tool for creating graphical models (UML) within QML, shipped with Qtronic. The QML graphical notation, created using Qtronic Modeler, was used for describing finite state machines along with the QML textual notation. Creating state machines in the QML graphical notation defines the state machine execution logic, which otherwise has to be defined in the textual notation. The very minimum of textual notation is a QML class corresponding to the graphically defined state machine and a system block describing the interfaces and possible messages of each interface. Graphical state machines in QML consist of states and transitions, as well as initial and final states. A state machine may also include an internal state machine, which is bound to a particular state. Furthermore, as illustrated in this project, it is possible to make use of a separate QML model by starting a state machine in

the textual notation, thus also including logic from synchronizing the logic within the model using a separate QML model. The transition logic of a state machine is described using transition strings. The transitions strings consist of triggers, guards and actions, although none is required.

Qtronic Modeler is a simple drawing tool. The project showed that the tool provides sufficient means for defining finite state machine logic, where the basis was the graphical notation which made use of the textual notation and class methods. The Qtronic User Manual [8] includes descriptions of QML and discusses differences compared to Java. However, the user manual does not include a complete description and an API (Application Programming Interface), but rather discusses the basic features in detail. An API would have been really useful instead of reading descriptions, which in some cases were not clear.

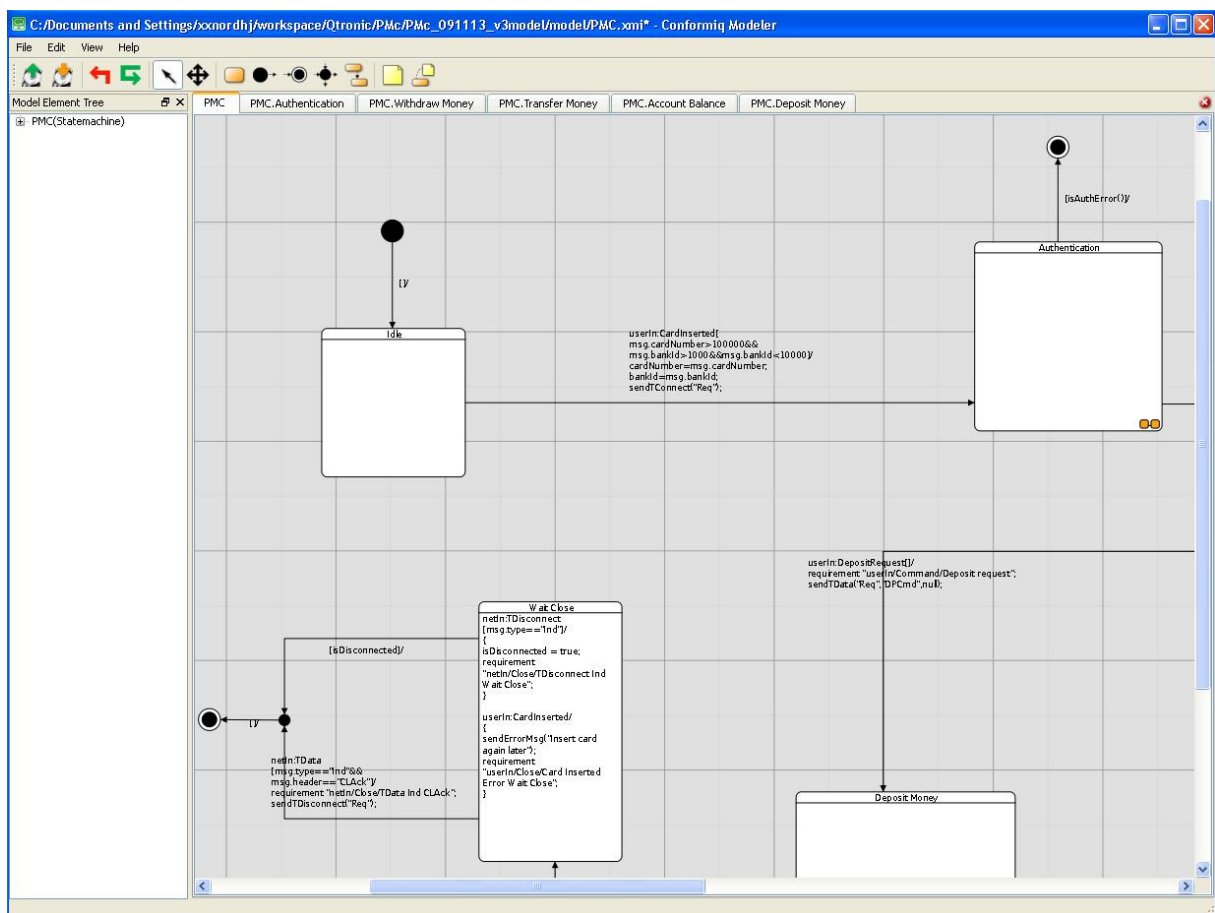


Figure 4.2: Qtronic Modeler

Qtronic Modeler was a simple drawing tool and not the most accurate tool for drawing state machines, especially not for transitions. The tool provided sufficient means for drawing states and transitions along with other features (such as internal transitions). However, when the number of transitions increased it was harder to keep a well-structured layout and graphical representation. For example, the placement of the transition strings was an issue. When the code blocks of the transitions strings increased in size and the number of transitions increased the solution was to move states and transitions further apart, which required one to zoom in and out as well as scroll horizontally and vertically within the tool. Given a large model this may have been expected, but not for this relatively simple model. The conclusion

regarding Qtronic Modeler is that it is a sufficient tool for describing state machine logic, but not an optimal tool. The conclusion regarding QML is that it included features that could be expected by an object oriented language, but that the Qtronic User Manual could have been more precise and included an API, although the standard library of QML is very limited to the standard library of Java.

5 Conclusion

In this thesis the concept of model-based testing (MBT) has been evaluated and tested in a case study and four experiments. In this chapter the principal goals are repeated and the corresponding results of the project discussed. Furthermore, this chapter includes suggestions of subjects for future work.

5.1 Results

The general goal of this thesis work was to evaluate the concept of MBT. This general goal included applying the concept by using a tool suited for this purpose on an existing test object and to successfully execute the tests. This was done by developing a model for a system and then gradually increased the complexity. During the study the process required to successfully execute tests was evaluated in terms of the concept and the MBT tool.

The general goal of the thesis work was further divided into sub-goals.

1. Learn the concepts of model-based testing
2. Learn to use a tool (Qtronic) for modeling, test code generation and test execution
3. Develop a framework for a finite state machine and implement a logic on the framework (with incrementally increased complexity) to be used as a test object.
4. Develop a model for the test object in the test tool and try different criteria for generation of test code
5. Establish the test environment including development of “glue” between the test code and the test object.
6. Execute the tests and incrementally make the test object and the model more complex.
7. Evaluate the result, document the experience gained and make recommendations.

The initial task of this project was to learn about the concepts of MBT. This was done in the pre-study, which initially focused on software testing in general, classic testing processes as well as testing at Tieto. The pre-study resulted in insights as to what MBT is, from where it originated, its scope, the process involved as well as the benefits and limitations of the method.

An important part of the project was to learn Qtronic, the MBT tool used in this thesis. A course in Qtronic was attended before starting the experimental part of the project. This course was held by Conformiq, the tool manufacturer, and lasted for three full working days. The course was a necessary step of the learning phase for using Qtronic, although tool documentation was available prior to the course. The Conformiq instructors provided useful information and shared experiences that could not be found in the Qtronic User Manual. Furthermore, one of the instructors stayed for two additional days following the course to assist and provide feedback for the initial modeling of the SUT. However, the learning phase continued after the course and also after the instructor had left. At this point in the learning phase, when basic functionality and features of Qtronic were known, the Qtronic User Manual provided useful information about details in Qtronic. The principal goal states that a tool for

modeling, test code generation and test execution was to be learned. Qtronic encompasses the first two mentioned features, but not the test execution feature. The version of Qtronic used in this project is a tool for offline generation of test scripts, meaning that test code is generated separately and it is then up to the tester to use the generated code. Hence, Qtronic did not include test execution of the generated test code.

The third principal goal of the project was to develop and implement a test object to be tested. This was achieved by developing a client-side protocol module for a client-server of an ATM system. This test object was implemented in Java and was extended throughout the project. Although the implementation followed a design pattern, namely the state design pattern, the implementation of the test object was not the main focus of the project. The goal was to develop a test object that was to be tested but the test object development was not the primary focus of the project.

When the concepts of MBT were known, the basics of the Qtronic tool were known and a test object was implemented the MBT process was started. The goal was to create a working model within Qtronic and try different criteria for generating test code. The modeling phases of the project were successfully completed, although the initial work required a learning phase and required more time to complete. Design factors were found to be important considerations as the model were gradually increasing in complexity. Although design factors were considered when starting modeling new possibilities and practices for design factors were discovered throughout the project.

Initially different criteria for generating test code were used to investigate the possibilities within Qtronic. The available criteria met the expectations for black-box testing and testing of finite state machines. Moreover, the different criteria were well defined in the Qtronic User Manual. Hence, the criteria for test generation were not evaluated and compared any further since the criteria used were sufficient to exercise the complete model.

A test environment was established, defining how the generated test code and the test object interacted. The test environment, or glue code, included a general test execution environment and script implementation dependent of the model. The test execution environment was defined at the beginning of the project and was then reused throughout the project. This test execution environment defined how the test code and the test object communicated, i.e. how the test code sent input and retrieved output from the test object. The test script (the test harness) was dependent on the model and contained the actual implementation for sending input and retrieving output from the test object. Thus, the test script made use of the test execution environment in its implementation.

Another goal was to execute tests against the test object and to incrementally make the test object and the model more complex. The generated test code was executed when the script implementation was completed. The test execution resulted in failures, due to errors in the model, the test script implementation or in the test object. As the errors were corrected the test execution eventually resulted in that all test cases passed. The test object, and subsequently the model, was incrementally extended given a successful test execution.

The final goal of the project was to evaluate the results and document experiences and recommendations. The results of the project work were evaluated in a qualitative analysis, which made use of metrics of the different artifacts as indications for the work effort required and to illustrate the gain of MBT. The modeling time and script implementation time, which

were the most time-consuming tasks, in relationship to the test code generated illustrated this gain. An important consideration was that the initial project work included a learning phase for all tasks involved in the process, from the modeling task in Qtronic to the execution of the generated test code. The analysis outlined results, considerations and implications for the different tasks of the project.

The key point of the analysis was the modeling of the project. The modeling was the most challenging and important task of MBT since the model is used for generating test code. The aspects documented, including design issues and design considerations are the most scalable and the most important aspects of the project work. However, the results in terms of modeling time and test generation are not scalable since they are dependent on the complexity of the model. Results and experiences for the script implementation were not scalable since this is dependent on the complexity of the model, the complexity of the test object and the test execution environment used.

5.2 Discussion

The thesis work was a feasibility study for the concept of model-based testing and the model-based testing tool Qtronic. The project included the creation of a test object, which was incrementally developed, and testing this test object using Qtronic.

Developing the test object during the project had its advantages. Since the test object was of relatively low complexity the model used for test generation was also of relatively low complexity. Hence modifications of the model were easier to trace through the different tasks involved in the complete testing process. The relationship between the model, modifications and the different tasks thus became more apparent compared with a scenario where a more complex system had been used. Furthermore, a significantly more complex system would require more time to get a complete and working first version of the model and less time to experiment with the test object and the model. The time frame of the project was also limited considering the time of the Qtronic course. Nevertheless, the project proved the concept and illustrated the gain of MBT. The project proved MBT to be a valid approach although the scalability of the project may be open to discussion. However, the design issues and design considerations for the modeling task should be more general since issues regarding design factors were encountered already in a model of relatively low complexity. There is no reason to believe that such issues would not be encountered for a more complex system, rather the opposite.

The potential use of a more complex system, perhaps an existing system at Tieto, may have resulted in results such as test generation times and modeling time that would have been directly applicable to current projects within the organization. However, a test object of a relatively low complexity resulted in more time to evaluate and trace the relationships between the different tasks and between different model versions. Using a more complex system, considering the learning phase, would have resulted in that significantly more time had been spent on modeling the initial system and to implement the test scripts necessary to interact with the SUT, and less time for experiments.

The most important experience of this project was the importance and the problems of the modeling task. In the MBT approach the focus is moved from test design and script implementation, compared to traditional testing approaches, such as the script-based testing

approach. The change of focus requires different tester skills as the abstraction level is increased. This is an important conclusion of this project since testing organizations and testers not always have the experience of object-oriented design and development in general. Thus this is an important consideration if MBT would be adopted in an organization.

In the MBT approach the model is the most important and crucial artifact. Thus awareness and considerations of the model quality will be extremely important. Considering large and complex systems that have the potential of being widely used and last for a long time, the testing will be of great importance. Such systems often evolve as the time goes by and often imply large maintenance duties, as well as incremental development through the development process. In such scenarios the quality of the model will be a crucial issue. Thus design factors and good modeling practices have to be applied from the very start of projects testing activities. Moreover, the model also has the potential of serving as an effective means of communication, which is a factor that should not be underestimated. Instead of reusing, extending and modifying test scripts, that often involve different testers through time, a well-structured model has the potential of make the whole testing process more effective.

Another important consideration and aspect of the MBT approach is the ability to reuse the model in a series of test generations. Given a complete and working model of the SUT, a large number of test generations may be performed, depending on the testing goals of each generation. Considering a complex system, even larger numbers of test cases may be generated from model. This may be an advantage, if the test generation times are not crucial. However, MBT has the advantage that the test generation is dependent on the testing criteria (test design configuration in Qtronic). Thus test cases for a particular feature of the system or a particular part of the model may be generated. Furthermore, the model does not need to incorporate the whole SUT, but only the parts of interest according to the testing goals. The latter may be a good solution if the complete system is too large to model or if the test generation takes too long time. Hence, the MBT approach offers flexibility to the software testing process since it includes one artifact to maintain and develop, which however results in strong requirements on the model quality.

To summarize, this project applied and evaluated the concept of MBT. The feasibility study illustrated the gain and the characteristics of the concept. The results and the experience indicate that MBT should be applied or further evaluated at Tieto. The conclusions of this project are listed below.

Benefits of the model-based testing approach:

- Automates generation of test code and test result evaluation
- Increases the abstraction level of testing
- A good model may serve as a means of communication in the development process
- Supports incremental development

Limitations on the project:

- The scalability of the project and the test object used in the feasibility study is an issue

5.3 Future work

To further evaluate the concept of MBT and Qtronic the following areas are proposed for future work.

5.3.1 Manual test design vs. MBT

To further evaluate the gain of MBT the concept could be compared to the currently applied manual test design. That is, MBT and manual test design could be applied to the same project or the same system. In such a project test design time, test coverage and discovered defects may be suitable metrics to compare the two approaches. Such a project should preferably be carried out simultaneously by two independent testers or test teams with corresponding experience. The reason for this is that performing one of the testing approaches before the other may affect the results of the following approach since knowledge about encountered defects in the system may be gained. For example, if MBT were applied first and resulted in a number of errors the knowledge of the encountered errors could be used in the evaluation of manual test design.

An important consideration for such a project would be the learning phase. If experienced testers perform the manual test design approach, the testers performing MBT should also be experienced. Difference in experience would impact the findings and the results between the different approaches. Furthermore, for the results of such an experiment to be applicable and scalable a good practice would be to use an existing system from the organization. The goal of the project would be to evaluate if MBT is worth the effort, and thus the results need to be scalable and applicable to the projects and the organization at large.

5.3.2 Modeling of current applications at Tieto

An alternative to the quantitative analysis mentioned above would be to apply MBT to an existing system or an existing project to perform a qualitative analysis. Such a project would be similar to this thesis but result in more scalable results and experience that in general could be applicable to the organization. An important consideration for such a project would be the learning phase. If a significant learning phase is required in the project this would have an impact on the results and the experiences. The reason for this is that if there is not enough knowledge from the start of the project, the initial modeling effort of the project may result in discoveries and problems later on in the project. Mistakes or lack of knowledge in the early phases of the project may result in problems later on that affect the success of the project. For example, if design factors were not considered when starting to model a complex system the lack of initial knowledge may have consequences for the later stages of the project.

5.3.3 Evaluation of the tool

A third project could be an evaluation of Qtronic, or other MBT tools. Evaluating a tool does not necessarily have to be a separate project, but could be included in the previously suggested projects. However, evaluating the capabilities of a specific tool, or comparing several tools, might be more easily performed in a separate project. The focus of such a project could be on test generation times and test coverage to gain more knowledge about the capabilities of specific tools before applying them in projects within the organization. The result of the project could outline expectations for modeling time, test generation time and test coverage. Such indications could be useful for time and cost estimations if MBT would be applied in current projects and to know what to expect in such a scenario as well as to minimize unexpected outcomes.

5.4 Final Comments

The project started with a pre-study of the MBT concept and an introductory course of the MBT tool Qtronic. Following the course a feasibility study of the concept was performed. The feasibility included the development of a test object, which was limited to the protocol module for the client of a simplified model for an ATM (Automated Teller Machine) client-server system. The feasibility study included four experiments, with different purposes and goals. The test object was incrementally developed during these experiments to simulate the process of applying MBT in a new project. The general goals of the project were to evaluate the MBT concept, the MBT tool Qtronic, to execute tests on a test object of incrementally increased complexity and to document experiences and make recommendations.

The project was a success since it proved and illustrated the concept of MBT as well as its gain. All four experiments resulted in successful test executions, where each test case automatically was assigned a pass/fail verdict and discovered errors in the test object were corrected. The main task of the project was the modeling task and the most important experience of the project is the importance of applying a good and well-structured design to the model. Design issues are important considerations since the model is the central artifact of this approach. Another important aspect and consideration is the learning phase required, since MBT increases the abstraction level and requires different skills compared to traditional software testing. This project has shown the gain and the characteristics of MBT and it is recommended to further evaluate this approach within the organization at Tieto.

References

- [1] Alain Abran and James W. Moore. Software Engineering Body of Knowledge (SWEBOK). IEEE, 2004.
- [2] Larry Apfelbaum and John Doyle. Model Based Testing. *Software Quality Week Conference*, 1997. Available: http://www.geocities.com/model_based_testing/sqw97.pdf.
- [3] Kent Beck. eXtreme Programming explained. Addison-Wesley, 2004.
- [4] Eddy Bernard, Bruno Legeard, Xavier Luck, Fabien Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. 2004. Available: <http://lfc.univ-fcomte.fr/publications/pub/2004/RR2004-16.pdf>.
- [5] Mark Blackburn, Robert Busser, Aaron Nauman. Why Model-Based Test Automation is Different and What You Should Know to Get Started. Software Productivity Consortium, 2004. Available: http://www.psqtcconference.com/2004east/tracks/Tuesday/PSTT_2004_blackburn.pdf.
- [6] Ilene Burnstein. Practical Software Testing. Springer, 2003.
- [7] Conformiq. <http://www.conformiq.com> (Acc 2009-11-04).
- [8] Conformiq. Qtronic User Manual. Available: <http://www.conformiq.com/downloads/Qtronic2xManual.pdf>
- [9] Lee Copeland. A Practitioner's Guide to Software Test Design. Artech House, 2004.
- [10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton. Model-Based Testing in Practice. *Proceedings of ICSE'99 (ACM Press)*, 1999. Available: <http://aetgweb.argreenhouse.com/papers/1999-icse.pdf>.
- [11] Ibrahim K. El-Far. Enjoying the Perks of Model-Based Testing. *STARWEST*, 2001. Available: http://www.geocities.com/model_based_testing/perks_paper.pdf.
- [12] Ibrahim K. El-Far, James A. Whittaker. Model-Based Software Testing. *Encyclopedia on Software Engineering*, Wiley, 2001. Available: http://www.geocities.com/model_based_testing/ModelBasedSoftwareTesting.pdf.
- [13] E. Farchi, A. Hartman, S. S. Pinter. Using a Model-Based Test Generator to Test for Standard Conformance. *IBM Systems Journal*, 2002.
- [14] Stuart R. Faulk. Software Requirements: A Tutorial. University of Maryland, 2004. Available: http://www.cs.umd.edu/class/spring2004/cmsc838p/Requirements/Faulk_Req_Tut.pdf
- [15] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison-Wesley, 1995.
- [16] Bogumila Hnatkowska. Verification of Good Design Style of UML Models. Institute of Applied Informatics, Wroclaw University of Technology, 2007. Available: <http://ftp1.de.freebsd.org/Publications/CEUR-WS/Vol-252/paper10.pdf>
- [17] Marnie L. Hutcheson. Software Testing Fundamentals. Wiley, 2003.

- [18] IEEE. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Institute of Electrical and Electronics Engineers Inc: New York, 1990.
- [19] IEEE. IEEE Standard for a Software Quality Metrics Methodology (IEEE Std 1061-1992). Institute of Electrical and Electronics Engineers Inc: New York, 1993.
- [20] ITEA. The DESS Methodology, Deliverable D1 Version 01 – Public. ITEA, 2001.
- [21] Cem Kaner, James Bach, Bret Pettichord. Lessons Learned in Software Testing. John Wiley & Sons Inc, 2001.
- [22] George R. Klare. The Measurement of Readability. Iowa State University Press, 1963.
- [23] Philippe Kruchten. The Rational Unified Process: An Introduction. Addison-Wesley, 2004.
- [24] John Lewis, William Loftus. Java Software Solutions: Foundations of Program Design. Addison-Wesley, 2001.
- [25] John D. McGregor. A Component Testing Method. Clemson University, 1997.
- [26] Ron Patton. Software Testing. Sams Publishing, 2001.
- [27] Martin Pol, Ruud Teunissen, Erik van Veenendaal. Software Testing: A Guide to the TMap Approach. Addison-Wesley, 2002.
- [28] A. Pretschner, W. Prenning, S. Wagner, C. Kühnle, M. Baumgartner, B. Sostawa, R. Zälch, T. Stauner. One evaluation of Model-Based Testing and its Automation. ISCE'05, ACM, 2005. Available: <http://www.inf.ethz.ch/personal/pretscha/papers/icse05.pdf>
- [29] Mark Priestly. Practical Object-Oriented Design With UML. McGraw-Hill Education, 2003.
- [30] Jonathan Rosenberg, et al. Session Initiation Protocol (RFC3261). Internet Engineering Task Force, 2002.
- [31] Torbjörn Ryber. Essential Software Test Design. Fearless Consulting, 2007.
- [32] Scrum Alliance. <http://www.scrumalliance.org> (Acc 2009-11-04).
- [33] Ian Sommerville. Software Engineering. Pearson, 2001.
- [34] Mark Utting. Position Paper: Model-Based Testing. The University of Waikato, 2005. Available: http://www.cs.waikato.ac.nz/~marku/papers/utting_mbt_position.pdf.
- [35] Mark Utting and Bruno Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2007.
- [36] Brent B. Welch, Ken Jones, Jeffrey Hobbs. Practical Programming in TCL and Tk. Pearson, 2003.
- [37] Wikipedia. Automated Teller Machine (ATM). Available: http://en.wikipedia.org/wiki/Automated_teller_machine (Acc 2009-11-04)

A Requirements and Specifications

In this appendix different types of requirements and specifications are discussed

A.1 Requirements

This section describes different types of requirements.

A.1.1 User requirements

User requirements in this context are high-level abstract requirements, or are statements, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it operates. They should only specify the external behavior of the system. These requirements should target managers who do not have a detailed technical knowledge of the system [33].

A.1.2 System requirements

System requirements are detailed descriptions of what the system should do, where system services and constraints are described in detail. The document including system requirements is sometimes called a functional specification. It may serve as a contract between the system end user and the software developer. These requirements should target technical staff, and maybe project managers [33].

Furthermore, software system requirements are often classified as functional and non-functional requirements [33].

A.1.3 Functional requirements

Functional requirements are statements of services the system should provide, and how the system should react to particular inputs and how the system should behave in particular situations. Sometimes they may also state explicitly what the system should not do [33]. Functional requirements simply specify a function that a system or a component must be able to perform [18]. Functional requirements are sometimes known as capabilities [1], and typically refer to requirements defining the acceptable mappings between system input values and corresponding output values [14].

A.1.4 Non-functional requirements (quality requirements)

Non-functional requirements are constraints on the services or functions offered by the system [33]. Non-functional requirements are sometimes known as constraints or quality requirements [1].

Quality requirements may be grouped in six headings. The quality requirements contain so many aspects that it is appropriate to categorize them [31]. These are briefly described below.

Functionality

Functionality regards whether the desired functions are present in a system or not. Functionality can be further divided into the following characteristics: suitability, correctness, compatibility, compliance with standards and security [31].

Reliability

This category is regarded with robustness measures. Reliability answers questions like “Is the system robust?”, and “Does it work in different situations?” It can be further divided into maturity, defect tolerance, restart (after defect) and accessibility [31].

Usability

Usability describes if the system is intuitive, comprehensible and simple to use [31].

Usability is the capability of the software product to be understood, learned, used and attractive to the user under specified conditions, and may be divided into understandability, learnability, operability, attractiveness and usability compliance [27].

Understandability is the capability of the software product to allow the user to understand whether the software is suitable and how it can be used for particular tasks. Learnability is the capability of the software product to allow the user to learn its application. Operability is the capability of the software product to allow the user to operate and control it. Attractiveness is the capability of the software product to be attractive to the user. Usability compliance is the capability of the software product to conform to standards, conventions, style guides or regulations to usability [27].

Efficiency

Efficiency regards how well the system uses resources. This includes time aspects, such as performance characteristics. It also includes resource requirements (for example scalability) [31].

Maintainability

The fifth category of quality requirements is maintainability. This category describes if the workforce, developers and users can upgrade the system, and how easy [31]. IEEE [19] defines maintainability as an attribute that relates to the amount of effort needed to make changes in the software. Maintainability may be sub-divided into quality factors: testability, correctability and expandability.

Testability is described as an indication of the degree of testing effort required. Correctability is described as the degree of effort required to correct errors in the software and to handle user complaints. Expandability is the degree of effort required to improve or modify the efficiency or functions of the software [19].

These quality factors have measures associated with them. For example, testability may be measured in statement coverage, branch coverage and test plan completeness. Correctability may be measured in closure time (for a reported problem) and fault rate. Finally, expandability may be measured by change effort, change size, change rate and number of changes [19].

Portability

The final classification, portability, describes if and how well the system can work on different platforms (for example with different databases). Subcategories include adaptability, installation requirements, compliance with standards and replaceability [31].

A.2 Specifications

In this section software specifications are discussed.

A.2.1 Software requirements specification

The software requirements specification, or software requirements document, is the official statement of what is required of the system developers. It should include both user requirements for a system and a detailed specification of the system requirements. If there are a large number of requirements, the detailed system requirements may be presented as separate documents. Also, it should only specify the external behavior of the system [33].

Software requirements specifications establishes the basis for agreement between customers and contractors or suppliers on what the software is to do, as well as what it is not expected to do. For non-technical readers this document is often accompanied by a software requirements definition document [1].

Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks and schedules. Software requirements are often written in natural language, but in this specification they may be supplemented by formal or semi-formal descriptions. The general rule is that notations should be used which allow the requirements to be described as precisely as possible [1].

This specification is supposed to be of use for a variety of users. Customers may specify requirements and read them to check that they meet their needs. Managers may use the requirements specification to plan a bid for the system and to plan the system development process. The requirements may be used by engineers to understand what system is to be developed. Test engineers may use the requirements to develop validation tests for the system, while maintenance engineers may use the requirements to help understand the system and the relationships between its parts [33].

A.2.2 System requirements specification

In the system requirements specification the system requirements are separated [1]. This specification describes the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. An example of such detail could be interfaces to other systems. This specification is really a part of the software requirements specification [33].

A.2.3 Software design specification

A software design specification is an abstract description of the software design which is a basis for more detailed design and implementation. This specification adds further detail to the system requirements specification, and is an implementation-oriented document which should be written for the software engineers who will develop the system [33].

A.2.4 Component specification

A comprehensive component specification should include three types of properties: operations, the object state and interactions. Individual operations should be specified in terms of constraints on their inputs and outputs, as pre and post-conditions. The state constraints of the object should be described by specifying invariants which depicts the limits on each of the attributes of the object. State-transitions diagrams could be used to define specific sequences

of operations that represent the object's protocols. Finally, the interactions among methods and attributes should be specified by a series of functional models, for example using object interaction diagrams. This constrains how the methods interact with each other either directly or indirectly through the components attributes [25].

A.2.5 Interface specification

Many software systems have to operate with other systems which already have been implemented and installed in an environment or are under development. The interfaces of the existing systems must be precisely specified if the new system and the existing systems must work together. These specifications should be defined early in the process and be a part of the software requirements specification. These specifications may include procedural interfaces, data structures and representation of data, if these are defined in existing sub-systems. These describe the data and operations that can be accessed through a sub-system interface [33].

A.2.6 Behavioral specification

Model-based specification is an alternative approach to formal specifications. In a behavioral specification the system specification is expressed as a system state model. System operations are specified by defining how they affect the state of the system model. In this way the behavior of the system is defined [33].

An example of such a model is a state chart diagram, or a state machine. In that case the model consists of a collection of states and transitions, and the relationship between them. The relationship consists of transitions between states, where transitions are triggered by events. Also, execution of transitions most often includes execution of corresponding actions. State machines could be quite more complicated, but could be used as described above in the simple case to model the behavior of a system. These state charts could be defined using UML, for example [29].

B Testing

This appendix contains further details and definitions of software testing.

B.1 Terminology

When discussing software testing fundamentals it is also important to introduce proper terminology to ensure that further discussion are based on a common vocabulary that is widely accepted in the academic world as well as in the industry.

B.1.1 Glossary

Errors

An error is a mistake, misconception, or misunderstanding on the part of a software developer [6].

Faults (or Defects)

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification [6].

Failures

A failure is the inability of a software system or component to perform its required functions within specified performance requirements [18].

System under test (SUT)

The system is the program, library, interface, or embedded system that is being tested [35].

Test Cases

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program part or to verify compliance with a specific requirement [18].

Test

A test is a group of related test cases, or a group of related test cases and test procedures (steps needed to carry out a test). A group of tests that are associated with a database, and are usually run together, is often referred to as a test suite [6].

Testbed

An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test [18].

Test Coverage

The degree to which a given test or set of tests addresses all specified requirements for a given system or component [18].

Test Harness

A test harness (or test driver) is a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results [18].

Test Objective

An identified set of software features to be measured under specified conditions by comparing actual behavior with required behavior described in the software documentation [18].

Test Oracle

A test oracle is a document or piece of software that allows testers to determine whether a test has been passed or failed [6].

Test Scripts

A test script is detailed instructions for the set-up, execution, and evaluation of results for a given test case [18].

Test Suite

A test suite is a collection of test cases [35].

Software Quality

IEEE Standard Glossary of Software Engineering Terminology gives two definitions of quality [18]:

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.
2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

B.1.2 Defects

This section describes and discusses concepts regarding defects.

Defect (or Fault)

A defect (fault) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification [6].

Defect root cause

A defect root cause is the origin, or source, of a defect. The sources of a defect can be education, communication, oversight, transcription, and process. Education involves the education of the software engineer, communication involves communication between software engineers, oversight involves software engineers omitting actions, transcription involves the software engineer knowing what to do but makes a mistake in doing it, and process involves a process misdirecting software engineers in his or her actions [6].

Defect detection

One goal of testing is to detect defects in the software. One way to do this is to designing test cases which tries to expose defects of the software. Defects can be classified in many ways, such as requirements and specification defects, design defects, coding defects and testing defects. Depending on the defect type of interest, different techniques may be used. These techniques are most commonly categorized as static or dynamic techniques [6].

Defect analysis and defect prevention

Defect analysis and defect prevention are two related processes. The goals for these processes are to analyze defects to find their root causes, take actions and make changes (both in the overall development process and in the testing process), as well as preventing defects from reoccurring [6].

Defect analysis involves the process of identifying the root causes of defects. This process aims to pinpoint the exact cause of defects so that actions can be taken to make improvements, both at an individual level and at a process level [6].

Defect prevention is the process that allows an organization to take actions to prevent defects from reoccurring knowing the root causes of defects. Activities in this process are action planning, action tracking, feedback, and process change [6].

Requirements and specification defects

The initial phase of the software life cycle is critical in terms of ensuring high quality in the software. Defects that originate in early phases can persist and be very difficult to remove later in the software development process. Requirements are often documented in natural language, thus they are often ambiguous, contradictory, unclear, redundant, and imprecise. The same applies for specifications since they also often are developed using natural language. Some specific requirements or specification defects are functional description defects, feature defects, feature interaction defects, and interface description defects [6].

Design defects

Design defects originate when components, interaction between components, interaction between components and external environments, or interactions with users are incorrectly designed. Design defects are related to coding defects, but when describing design defects a detailed design specification is assumed. If such a specification does not exist many of the design defects may be classified as coding defects instead [6].

Coding defects

Coding defects are the result of implementation errors in the code. Coding defects may come from a failure to understand programming language constructs and miscommunication between developers, or from transcription and omission origins [6].

Testing defects

Documentation related to testing may contain defects. Such documentation may be test plans, test harnesses, and test procedures. Reviews are most often used to detect defects in test plans [6].

B.2 Testing techniques

B.2.1 Static techniques

This section contains descriptions of common static test techniques.

Inspection

Inspection is the most formal review technique and is strictly governed. The inspection participants prepare themselves by examining selected areas according to role descriptions and check-lists. The inspection is then performed in terms of a meeting where all points of view are recorded and how they are to be addressed may be discussed. This way of working is formal, hence only a limited amount of material is examined on each occasion. For this type of review to be effective the participants have to be trained and the meeting itself have to be lead by an experienced moderator [31].

Walkthrough

A walkthrough is a simpler and less formal review technique, which involves the author presenting his or her material to a selected group of participants. The goal is to more quickly involve the participants in the test basis and to create a common picture rather than identifying defects [31].

Technical review

Technical reviews focus in the technical parts of the project, such as architecture and program design. Technical experts and architects and other developers are the primary participants. The purpose is to evaluate choices of solution and compliance with standards, as well as other documentation [31].

Informal review

Informal reviews include more than two people looking through a document or code that one of the participants has written. The purpose is to detect defects, but usually no check-lists are used and the result does not need to be documented [31].

B.2.2 Black-box techniques

This section describes commonly used black-box testing techniques. Black-box testing techniques are test-specification techniques that derive test cases from the externally visible properties of an object without having knowledge of the internal structure of this object [27].

Equivalence class partitioning

Equivalence class partitioning is a method for selecting test inputs when performing black-box testing. The approach results in a partitioning of the input domain of the system under test. The technique may also be used to partition the output domain, but this is not a common usage. For each resulting partition, or equivalence class, the tester selects a representative member of that class. The approach assumes that all members of an equivalence class are processed in an equivalent way by the software. Moreover, the tester should consider both valid and invalid equivalence classes, where invalid classes represent invalid or unexpected inputs [6].

Boundary value analysis

Referring to the equivalence class partitioning definition above, boundary value analysis covers elements close to the edges of equivalence classes instead of any element of the class as in equivalence class partitioning. The reason for choosing equivalence class members of input and output close to the upper and lower edges is that such test cases are often valuable

in revealing defects. For example, if a specification of a software module states that the input values have to lie in the range between -1.0 and 1.0, valid test cases should include values for ends of the range, and invalid test cases should include values just above and below the ends. The result would be test cases including input values of -1.0, -1.1, 1.0 and 1.1 [6].

State transition testing

State transition testing is based on the concept of finite state machines. In this approach the test case design makes use of an existing state transition diagram (a defined finite state machine). The state transition diagram is a model of the system to be tested, which is used when designing test cases. The test cases are designed to cover the states, transitions between states, and inputs and events that trigger state changes of the state machine [6].

Error guessing

The error guessing approach is a test design approach based on the testers', or developers', past experience with systems under test, and their intuitions as to where defects may be found. The tester or developer may be able to make an educated "guess" as to which types of defects that may be present and design test cases to reveal them [6].

B.2.3 White-box testing techniques

This section describes commonly used white-box testing techniques. White-box testing techniques are test-specification techniques that derive test cases from the internal properties of an object, with knowledge of the internal set-up of the object [27].

Statement testing

IEEE [18] defines statement testing as testing designed to execute each statement of a computer program, where a statement, within a programming language, is defined as: "A meaningful expression that defines data, specifies program actions, or direct the assembler or compiler [18]."

Branch testing

IEEE [18] defines branch testing as testing designed to execute each outcome of each decision point in a computer program. They further define a branch as a point in a computer program at which one of two or more alternative sets of program statements is selected for execution [18].

Path testing

IEEE [18] defines path testing as testing designed to execute all or selected paths through a computer program [18]. In the context of software engineering a path is defined as: "A sequence of instructions that may be performed in the execution of a computer program [18]."

Mutation testing

IEEE [18] defines mutation testing as a testing methodology involving execution of two or more program mutations using the same test cases to detect differences in the mutations [18]. They further define a program mutation as: "A computer program that has been purposely altered from the intended version to evaluate the ability of test cases to detect the alteration [18]."

Loop testing

The purpose of loop testing is to detect common defects associated with loops [6]. A loop is a sequence of computer program statements that is executed repeatedly until a condition is met or while a given conditions is true [18].

For example, using a simple loop with a range of zero to n iterations, test cases should be designed so that there are: zero iterations, one iteration, two iterations, k iterations (where $k < n$), $n - 1$ iterations and $n + 1$ iterations (if possible) [6].

B.3 Traceability matrix

Traceability is the ability to trace the connections between artifacts of the testing life cycle or software life cycle. It is the ability to track the relationship between test cases and the model, between the model and informal requirements, or between the test cases and the informal requirements [35]. A traceability matrix is a table that shows the relationships between two different artifacts of the testing life cycle. For example, the relationships between informal requirement identifiers and generated test cases [35].

In this thesis the traceability matrix will describe the relationship between coverage goals and generated test cases. Figure B.1 below is an example of such a matrix.

Testing Goals	1	2	3	4	5	6	7	8	9
Requirements									
13.2.2.4 2xx Responses									
UAC core establishes session with ACK		X		X	X		X		X
15.1 Terminating a session									
UAC core terminates a session by sending BYE					X		X		X
UAS core sends OK in response to BYE				X					
17.1.1.2 INVITE timers									
Resends INVITE after A timeout	X					X			
Terminates INVITE cycle after B timeout						X			
17.1.2.2 Non-INVITE timers									
Resends BYE after E timeout							X		X
Resends CANCEL after E timeout			X					X	
Terminates BYE cycle after F timeout									X
Terminates CANCEL cycle after F timeout								X	

Figure B.1 Example of a traceability matrix

This figure is a print-screen from the traceability matrix produced during test generation in a Qtronic [8]. The model used is an example that comes with Qtronic. In this case the coverage goal of the test generation is set to cover functional requirements. However, coverage goals may be states, transitions, branches, to mention a few.

The model demonstrates behavior of the client side of the SIP protocol (specified in RFC3261 [30]) on an abstract level. SIP is an application-layer control (signaling) protocol for creating, modifying and terminating sessions with one or more participants. Examples of such sessions include Internet telephone calls, multimedia distribution, and multimedia conferences. This SIP model describes partial functionality of a SIP User Agent Client, and includes call setup, call termination by caller or callee and call cancellation during call setup. The model also includes the timers associated with these functionalities [8].

The matrix specifies what coverage goals are covered for a given test case (ranging from 1 to 9). For example, test case 1 in the figure above covers only the “Resends INVITE after A timeout” requirement.

C Test System

This appendix describes the different parts of the test system in more detail, using finite state machines.

C.1 Specifications

This section includes the specifications in terms of UML state diagrams. These specifications were used to implement the test object as well as to create QML models in Qtronic. In these diagrams the error handling, such as handling of negative acknowledgements, are omitted since the diagrams would be significantly more complex.

C.1.1 Version 1

This section includes the initial specification which was used for the first experiment.

Top-level state machine

The figure below is a UML state diagram of the top-level state machine of the protocol module, where the states Authentication and Withdrawal have internal state machines.

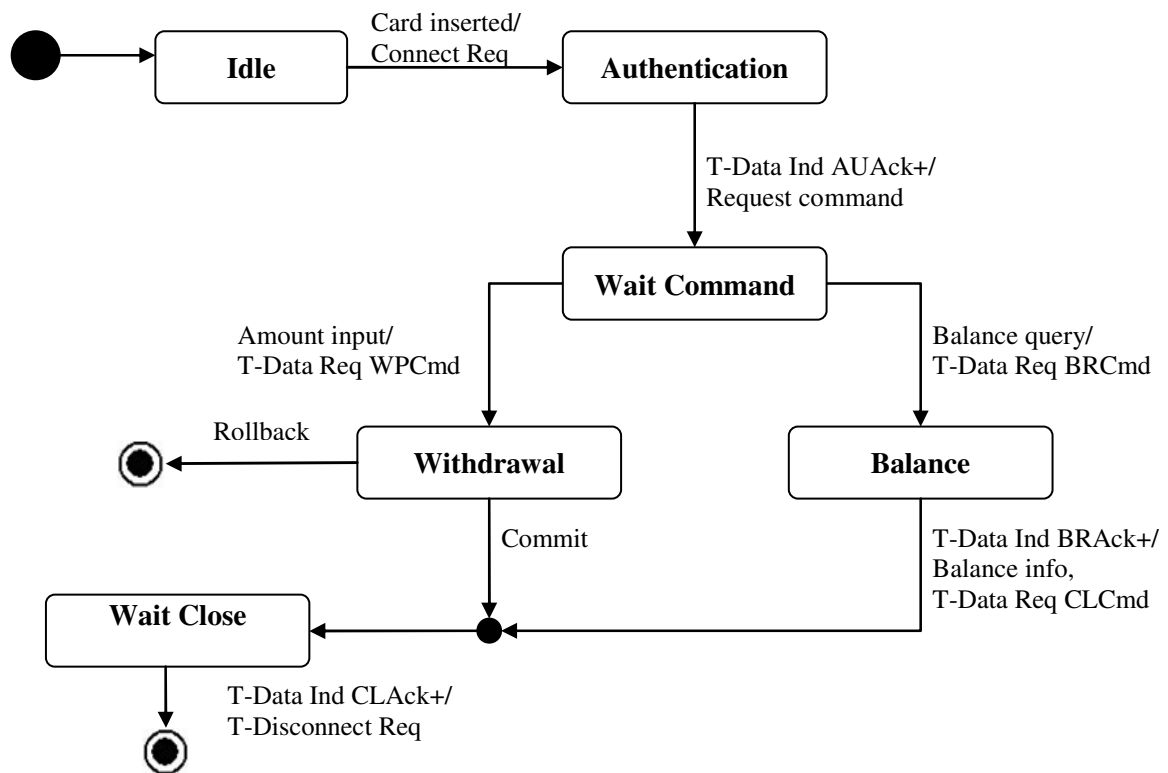


Figure C.2: Specification 1: Top-level state machine

Authentication state machine

The authentication state machine handles the user authentication, involving connection setup, card verification, pin code authentication and the corresponding network communication.

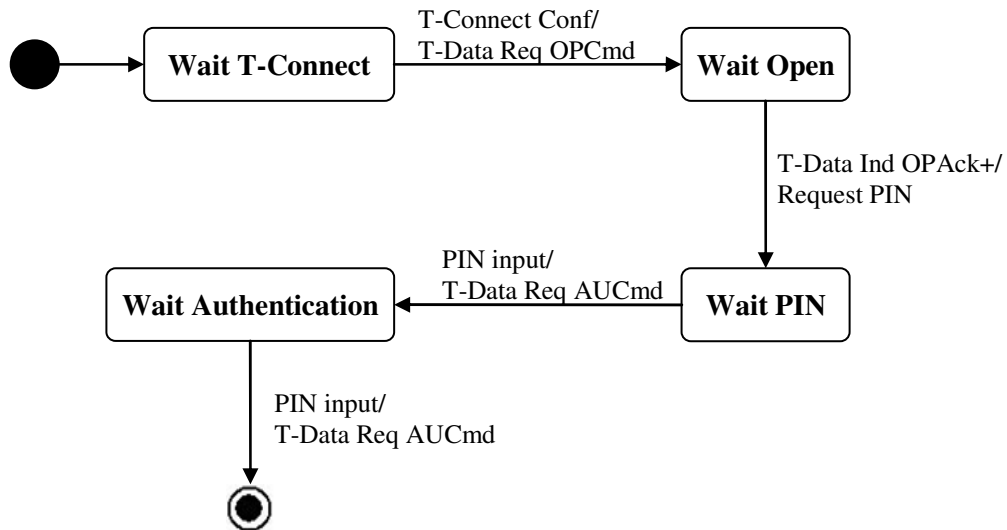


Figure C.3: Specification 1: Authentication state machine

Withdrawal state machine

The withdrawal state machine handles money withdrawal, which primarily includes amount verification against the account. It also includes verifying that the ATM machine (the user interface) has enough bills (dispense result), and functionality for either scenario.

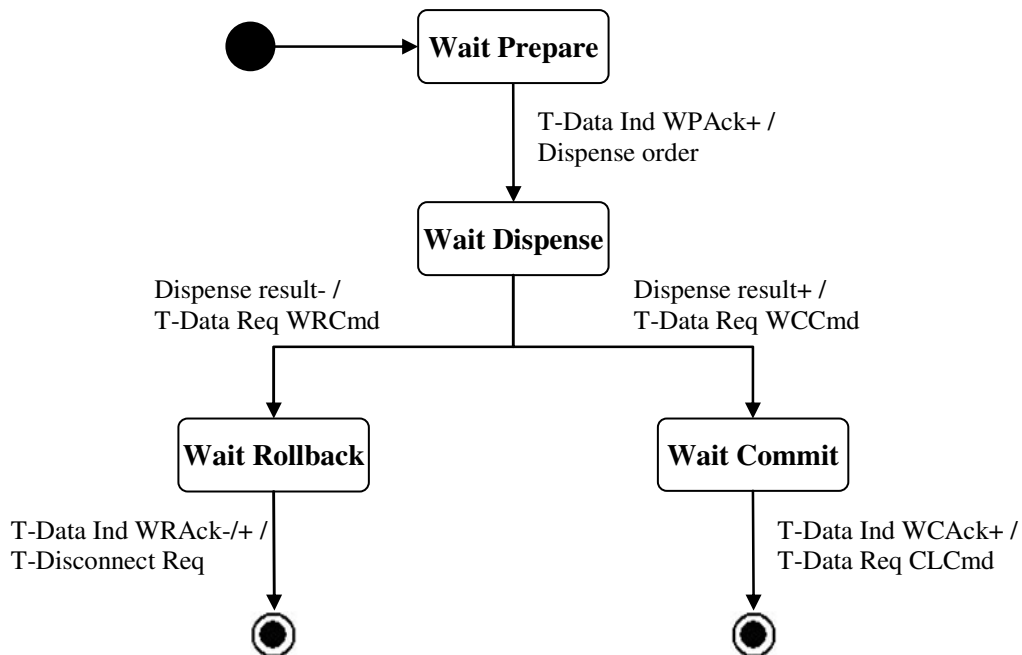


Figure C.4: Specification 1: Withdrawal state machine

C.1.2 Version 2

This section includes the specification used in the second experiment.

Top-level state machine

The figure below specifies the behavior of the top-level state machine of the protocol module. The states Withdrawal, Transfer and Deposit all have internal state machines. The withdrawal and authentication state machines are the same as in version 1.

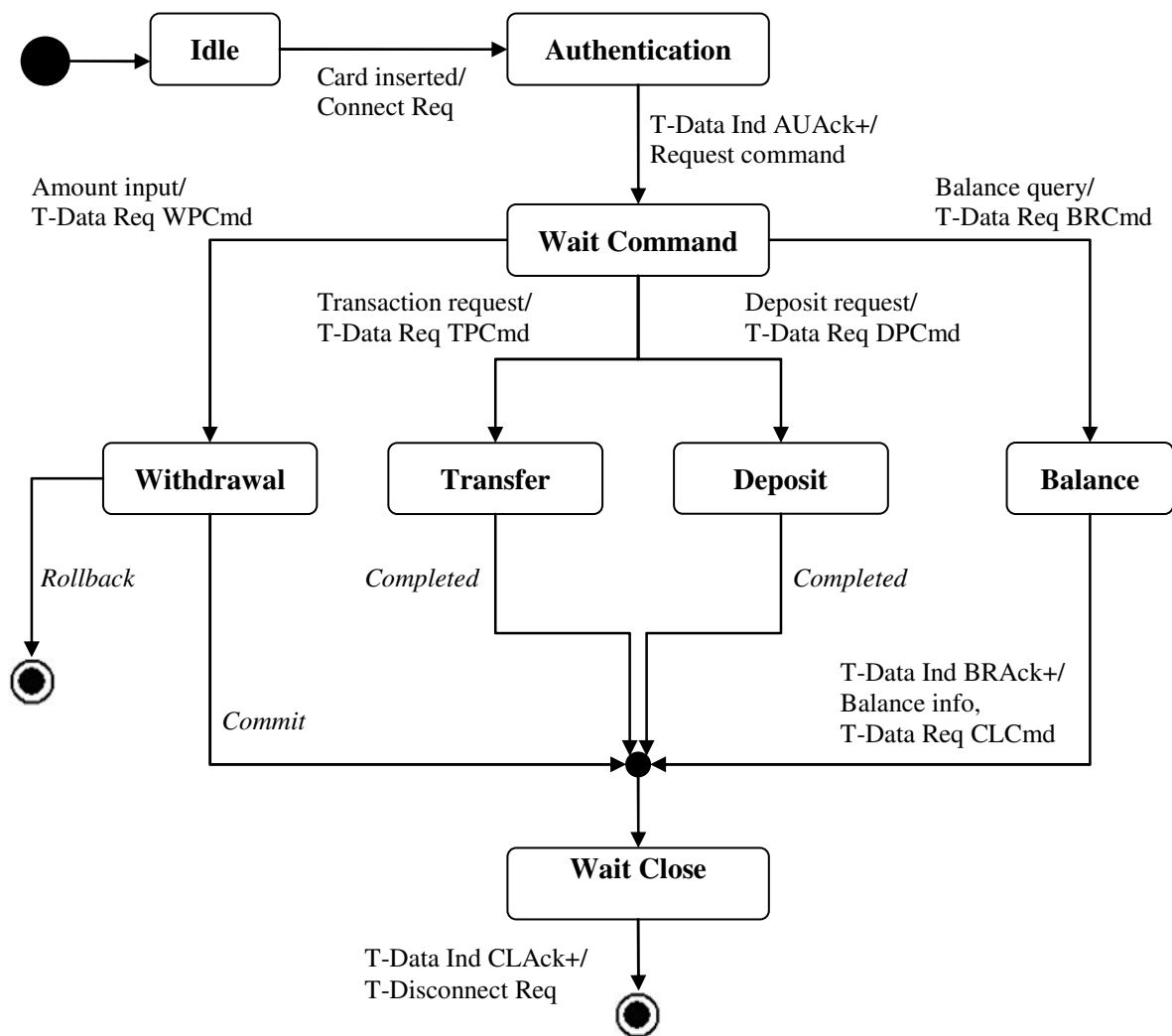


Figure C.5: Specification 2: Top-level state machine

Transfer state machine

The transfer state machine handles account transactions, which includes account and amount verifications against the server.

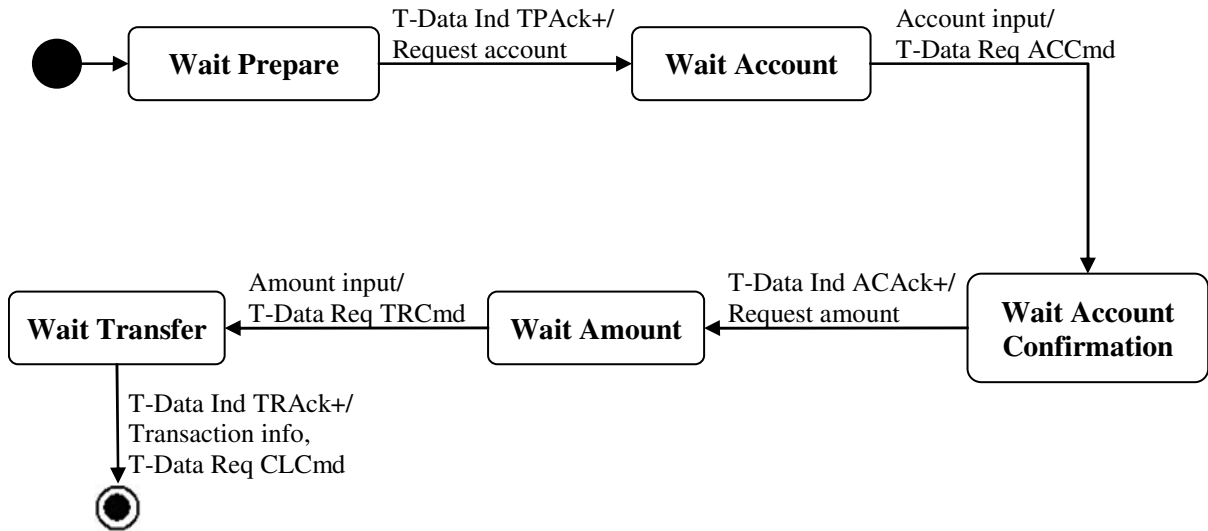


Figure C.6 : Specification 2: Transfer state machine

Deposit state machine

The deposit state machine handles account deposits, which includes the insertion of bills to the ATM and the creation of a receipt.

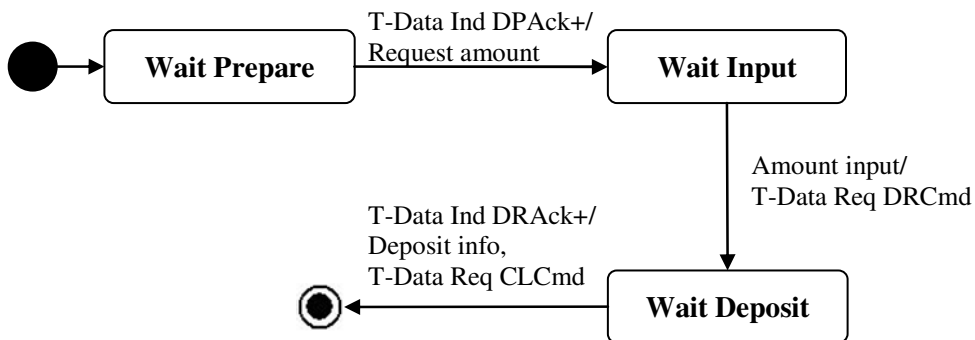


Figure C.7: Specification 2: Deposit state machine

C.1.3 Version 3

This section includes the specification used in the third experiment.

The top-level state machine is the same in this specification as in Specification 2. The difference is that a biometric authentication is required to complete requests to the server. This applies for withdrawal, deposit, transfer and balance requests. New for this specification compared to Specification 2 is that the Balance state of the top-level state machine includes an internal state machine. The biometric authentication is modeled as separate state machine which is used in the internal state machines. The biometric authentication process must be successfully completed for the requests to be processed.

Top-level state machine

The figure below specifies the behavior of the top-level state machine of the protocol module. The states Withdrawal, Transfer, Deposit and Balance all have internal state machines. The difference compared to version 2 is that these internal state machines include a biometric authentication.

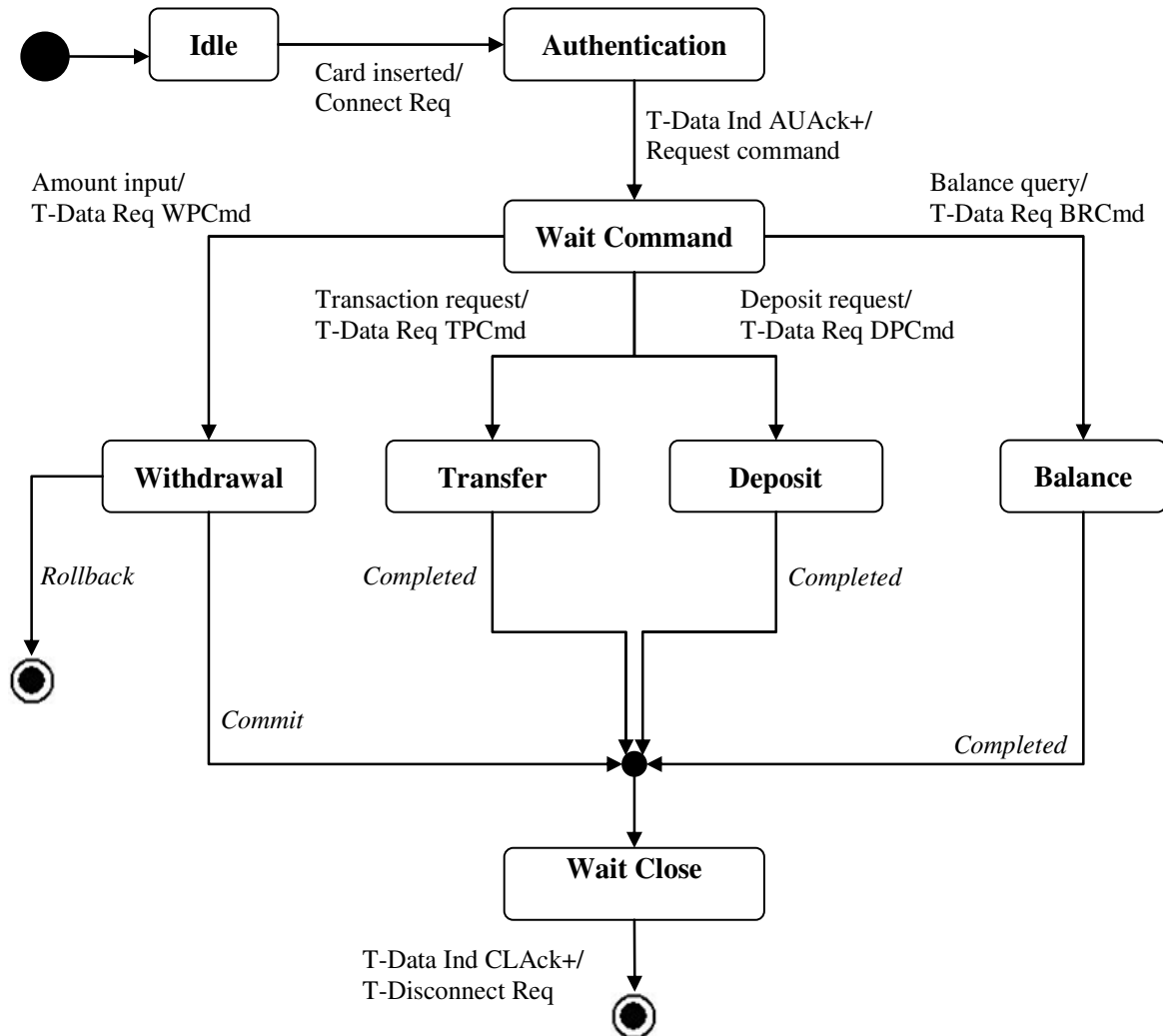


Figure C.8: Specification 3: Top-level state machine

Biometric authentication state machine

The biometric authentication state machine handles the biometric information input and verifies this against the server. The first step in the process is to receive a random seed number from the server, which together with the biometric information authenticates the request.

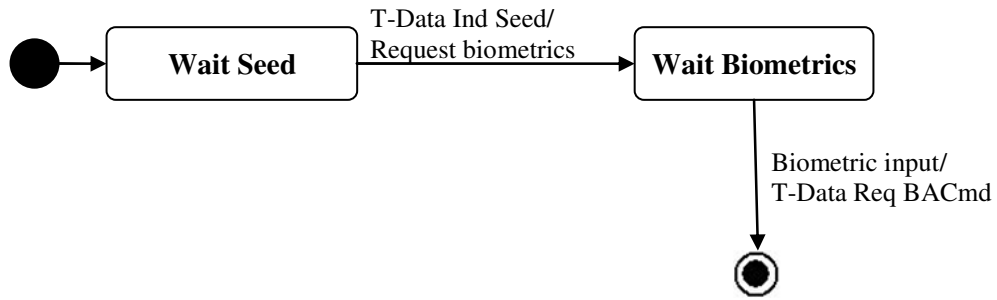


Figure C.9: Specification 3: Biometric authentication state machine

The introduction of this process yields changes in the withdrawal, deposit and transfer state machines as well for the Balance state in the top-level state machine.

Balance state machine

In this specification the Balance state include an internal state machine. This state machine handles the biometric authentication as described in the figure below. If the biometric authentication fails the request is terminated.

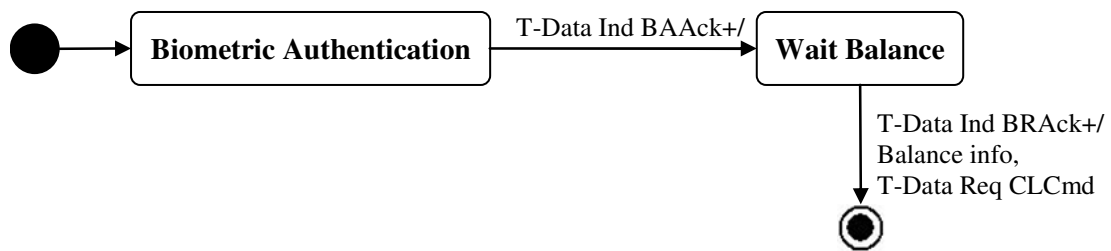


Figure C.10: Specification 3: Balance state machine

Withdrawal state machine

The withdrawal state machine of this version, compared to version 1, required a successful biometric authentication to carry out the account withdrawal request. The Biometric Authentication state in the state machine below includes an internal state machine to encapsulate that logic. If the biometric authentication fails the request is terminated.

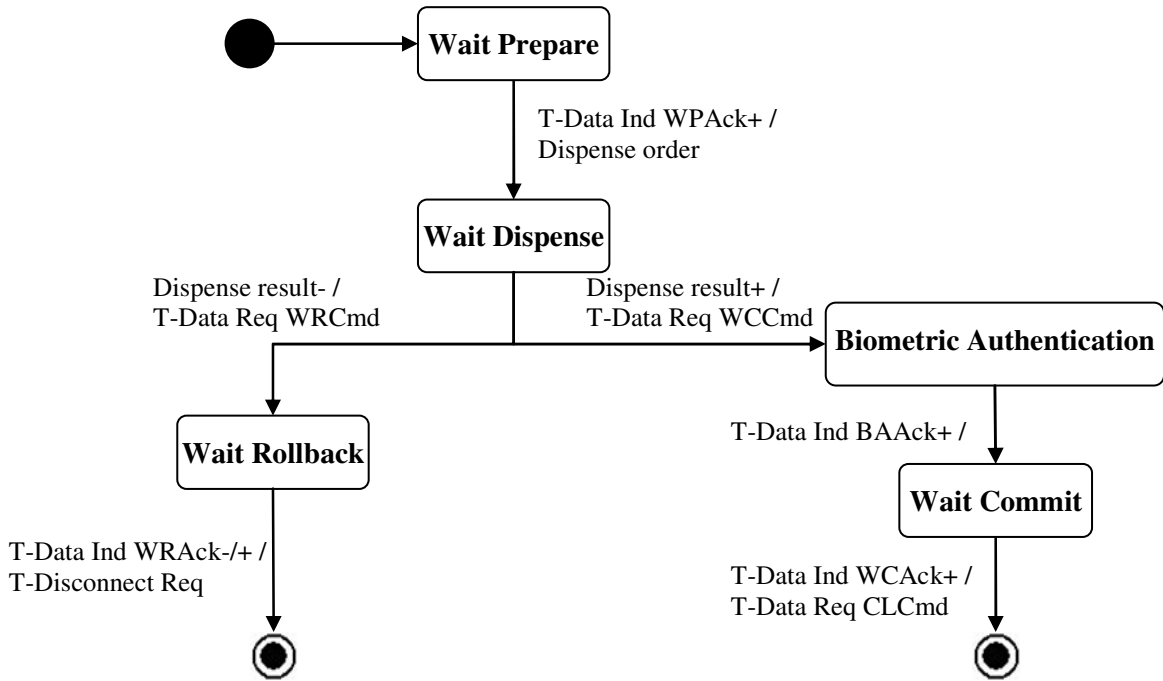


Figure C.11: Specification 3: Withdrawal state machine

Transfer state machine

As for the Withdrawal state machine, the Transfer state machine of this version required a biometric authentication to carry out account transaction requests. The Biometric Authentication state includes an internal state machine for describing the authentication process. If the biometric authentication fails the request is terminated.

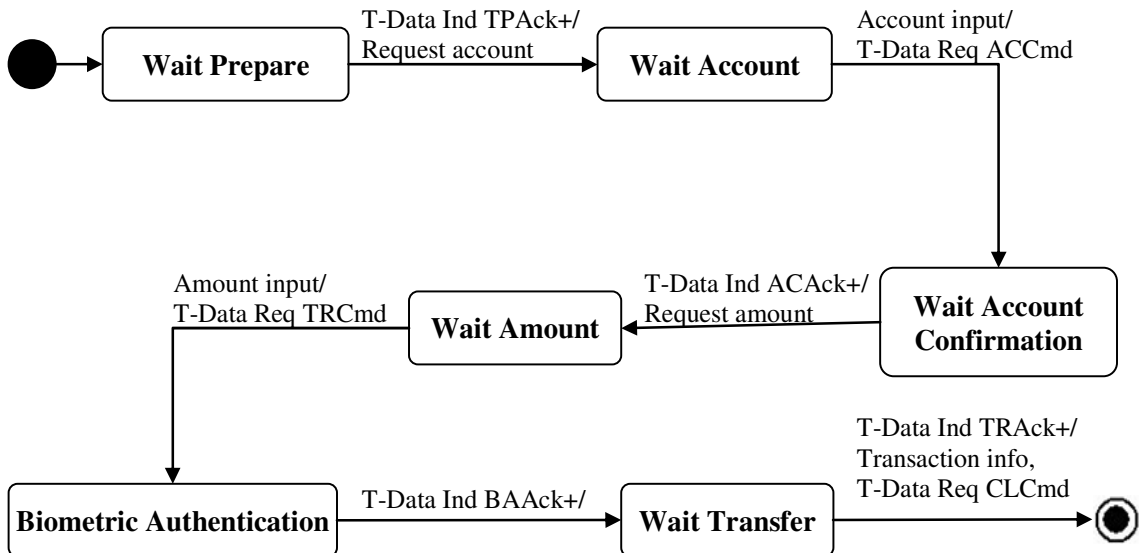


Figure C.12: Specification 3: Transfer state machine

Deposit state machine

As for the other state machines handling account requests the Deposit state machine requires a successful biometric authentication. If the biometric authentication fails the deposit request is terminated.

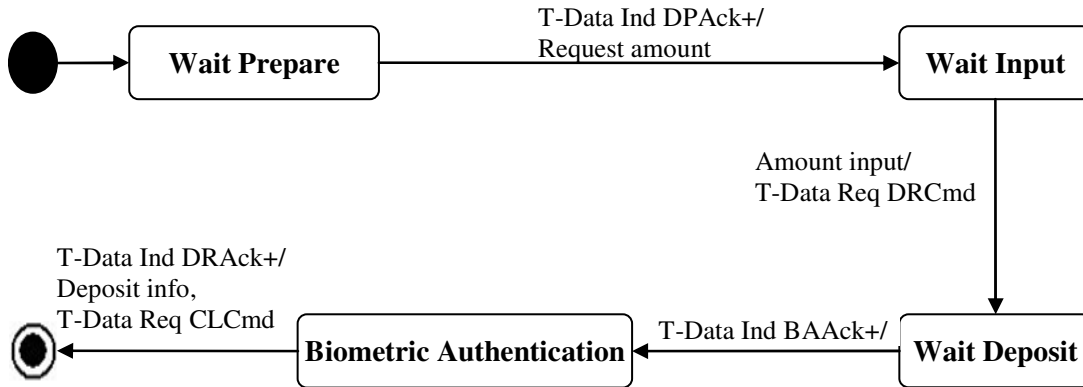


Figure C.13: Specification 3: Deposit state machine

C.2 Qtronic

This appendix describes features of Qtronic in greater detail.

C.2.1 Test design configuration parameters

The parameters of the test design configuration are grouped into requirements, control flow, conditional branching, state chart and dynamic coverage.

Requirements

Requirements may be established to establish additional test goals driven by functional requirements. The requirements, or requirement links, are marked in the model using requirement statement, which is a QML extension to Java. The argument of this statement is a constant string, really a name of the requirement. This requirement marks a point in the model which can be used as a testing goal to guide Qtronic in the test generation [8].

Control flow

The control flow parameters include testing and coverage goals for QML statements and methods used in the model. An example of statement coverage may be to cover an assign statement, assigning a value to a variable, and an example of method coverage may be to cover a function call in the model to a QML function in the model [8].

Conditional branching

Conditional branching includes boundary value analysis, branch coverage, and atomic condition coverage. Boundary value analysis covers the boundary value cases for all the arithmetic comparisons. This is a technique for determining tests covering known areas of frequent problems at the boundaries of input ranges. Branch coverage tells Qtronic to look for QML branches, such as then and else branches of if statements, at least once. Atomic

condition coverage tests every QML atomic condition branch, such as different sides of a Boolean “and” (i.e.. &&), at least once [8].

State chart

State chart parameters include state coverage, transition coverage, 2-transition coverage, and implicit consumption. State coverage guides Qtronic to generate test cases that cover every UML state at least once. Similarly, transition coverage guides Qtronic to generate test cases that cover every UML transition. 2-transition coverage tests every pair of two subsequent UML transitions at least once. Implicit consumption tests that the system correctly ignores messages that are not handled by any transition for a given state [8].

Dynamic Coverage

Dynamic coverage includes All Paths-States, All Paths-Transitions, and All Paths-Control Flow. All Paths-States tests every possible sequence of UML states at least once. All Paths-Transitions test every possible sequence of UML transitions at least once. All Paths-Control flow All Paths-Control Flow tests every possible sequence of conditional branches, such as then and else branches of if statements, at least once [8].

C.2.2 Test generation options

Test generation options are defined at a project-level in Qtronic. Thus they apply for all test design configurations defined for a specific project. This section briefly describes the available test generation options.

Lookahead Depth

This option controls the amount of lookahead for planning the test cases. The specified value, ranging from 1 to 7, corresponds to the number of external input events to the system or timeouts. If the logic in the design model manipulates the data after a certain number of external events the lookahead depth must be increased. The reason for this is that Qtronic must be aware of this to make decisions on the data values. However, if the lookahead value is high that will affect test generation times [8].

Maximum Delay

The maximum delay option defines the time interval, ranging from 0 seconds to 10 minutes, in which it is valid to deliver a message. The setting of this option depends on the application being tested [8].

Only Finalized Runs

If this option is selected Qtronic will only generate test cases ending in a “clean” state. Typical “clean” states are final states in the model. However, final states for internal state machines only indicate that the internal state machine execution read an end and not that the top-level state machine reached an end [8].

Require Conversion for Interoperability testing

When this option is selected, a require statement is handled as an assert statement whenever a thread received a message from another thread, or in a thread is awakened by a timeout when it is not waiting for any external interfaces. A require statement in QML requires a Boolean argument supplied to be true. A thread is a thread of execution in a program. A model in Qtronic may include several threads running concurrently, possibly communicating with each other [8].

OSI Methodology Support

Selecting this option activates the “OSI Methodology” feature of Qtronic. This feature provides support for generating test cases conforming to the OSI methodology for organizing test cases as described in ISO 0646-1 [8].

The feature divides all test cases into three sections: preamble, body, and postamble. Moreover, every test case is named according to by the name of one of the requirements verified in the body, except if the body does not verify any new requirements. In the latter case test cases are named according to one of the structural checkpoints of the body. The cases are also ordered in dependency order, so typically later test cases depend on earlier test cases [8].

Test Case Name Prefix

This option simply defines the default name prefix that is given to new test cases. The default value is “Test Case”, thus test cases are given names as “Test Case 1” [8].

C.2.3 Views for test generation result analysis

This section briefly describe the available views for analysis the generated test cases.

Traceability Matrix View

The traceability matrix correlates the coverage options to the individual test cases. For example, this view may display which states in the model are covered by a particular test case.

Coverage Editor

The coverage editor presents the final coverage percentage in regards to the coverage settings. For example, suppose that one want to cover all states of the model, and if successful, the coverage editor will display the number 100 for that coverage setting.

Test Case List

The test case list shows the generated test cases, including the test case names and the generation date.

Test Case View

The test case view shows the interaction between the tester and the SUT in a given test case. This view displays the input and the expected output of the SUT for a particular test case.

Test Step View

The test step view shows detailed information about the messages that are transferred between the tester and the SUT in a given test case. In effect, this is a more detailed view of the Test Case View.

Execution Trace View

The execution trace view links the test cases back to the model from which they were generated. This view shows a sequence of states through the model for a given test case.