Evaluation of Model Based Testing and Conformiq Qtronic

by

Bilal Khan & Song Shang

LIU-IDA/ LITH-EX-A- -09/049- - SE

2009-09-30
Final Thesis

Evaluation of Model Based Testing and Conformiq Qtronic

by

Bilal Khan & Song Shang

LIU-IDA/ LITH-EX-A- -09/049- - SE

2009-09-30

Supervisor & Examiner: Professor. Kristian Sandahl
We would like to thank our teacher, supervisor and examiner Professor Kristian Sandahl and Michael Lidén from Conformiq, for their direction, assistance and guidance through out in this thesis. We also like to thank our teachers of Software Engineering and Management program for their sharing their knowledge and experiences.

We would like to dedicate our thesis to our parents for their extreme support and prayers in accomplishing this goal.

Bilal Khan & Song Shang
Abstract

The Model Based Testing is one of the modern automated testing methodologies used to generate test suits automatically from the abstract behavioral or environmental models of the System Under Test (SUT). The Generated test cases are abstract like models, but these test cases can also be transformed to the different test scripts or language specific test cases to execute them. The Model based testing can be applied in different ways and it has several dimensions during implementation that can be changes with nature of the SUT. Since model based testing is directly related with models, the model based testing can be applied at early stages of development that helps in validation of both models and requirements that could save time of test development at later stages. With the automatic generation of test cases, requirements change is very easy to handle with the model based testing as it requires fewer changes in the models and reduces rework. It is also easy to generate a large number of test cases with full coverage criteria using the model based testing that was hard to produce with traditional testing methodologies. Testing non-functional requirements is one field in which the model based testing is lacking; quality related aspects of the SUT difficult to be tested with the model based testing.

The effectiveness and performance of model based testing is directly related to the efficiency of CASE tool that implementing it. A variety of CASE tools based on models are currently in use in different industries. The Qtronic tool is one generating test cases from abstract model of SUT automatically.

In this master thesis detailed evaluation of the Qtronic test case generation technique, generation time, coverage criterion and quality of test cases are analyzed by modeling the Session Initiating Protocol (SIP) & File Transfer Protocol (FTP). Also generation of test cases from models manually and by using the Qtronic Tool. In order to evaluate the Qtronic tool, detailed experiments and comparisons of manually generated test cases and test case generated by the Qtronic are conducted. The results of the case studies show the efficiency of the Qtronic over traditional manual test case generation in many aspects. We also show that the model based testing is not effective applied on every system under test, for some simple systems manual test case generation might be a good choice.
Table Of Contents

Acknowledgments ................................................................. i
Abstract ........................................................................ iii

Introduction ........................................................................ 1
  Purpose of thesis .................................................................. 1
  Research Method .................................................................. 1
  Efforts and Contributions .................................................. 2

1.1 Introduction of Model Based Testing .................................. 3
  1.1.1 Background ................................................................ 3
  1.1.2 What is model based testing ........................................ 4
  1.1.3 The process of model based testing .............................. 4
  1.1.4 Benefits and Limitations ............................................ 7

1.2 Dimensions of the Model Based Testing .............................. 9
  1.2.1 Subject of the Model .................................................. 10
  1.2.2 Model Redundancy .................................................... 11
  1.2.3 Model Characteristics .............................................. 12
  1.2.4 Model Paradigm ....................................................... 13
  1.2.5 Test Selection Criteria .............................................. 14
  1.2.6 Test Generation Technology ...................................... 15
  1.2.7 Online or Offline Testing ......................................... 16

1.3 Previous Research on MBT .................................................. 17

Chapter 2: Constructing The Model ......................................... 20
  2.1 Steps of modeling ........................................................ 20
  2.2 Unified Model Language ............................................... 21

Chapter 3: Conformiq Qtronic™ ........................................... 24
  3.1 The Conformiq Company ............................................... 24
  3.2 Conformiq Qtronic™ Introduction .................................. 24
  3.3 Automated Test Generation Flow with Qtronic .................. 24
  3.4 The Qtronic Coverage Criteria ....................................... 25
    a) Requirements Coverage ............................................ 26
    b) Transition Coverage ............................................... 26
    c) State Coverage ....................................................... 26
    d) 2-Transition Coverage ............................................ 26
    e) Control Flow Coverage ............................................ 26
    f) Atomic Condition Coverage ..................................... 26
    g) Boundary Value Analysis ......................................... 27
    h) Statement Coverage ............................................... 27
    i) Method Coverage .................................................... 27
    j) All Paths Coverage ................................................. 27
    k) Implicit Consumption .............................................. 27
  3.5 Lookahead Depth ......................................................... 27
Table OF Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The MBT Process (Mark Utting B. L., 2007)</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>MBT Seven Dimensions. (Mark Utting A. P., 2006)</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>GUI Testing Frame Work (Xie, 2006)</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>State Machine Diagram (Fowler, 2003)</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Automated Test design Flow</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Qtronic Test Case Tester Interaction</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>Qtronic Test Case Steps</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>Time Comparison</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Example for the Path selection</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Example for the atomic condition selection</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>Test Case Generation Difference</td>
<td>46</td>
</tr>
</tbody>
</table>
Introduction

This thesis presents and discusses the effectiveness and efficiency of the MBT by evaluating the Qtronic tool that implements the MBT methodology. The Qtronic tool is currently in use in big industries like Ericsson and Nokia because of its powerful automatic test case generation technology from abstract models of SUT. This thesis mainly focuses on the test case generation technology of the Qtronic and quality of generated test cases in comparison with manual generation method.

Purpose of thesis

The main purpose of this thesis was evaluation of the MBT methodology in general and to experiment practical case studies to evaluate the Conformiq Qtronic tool. The main evaluation aspects of the Qtronic tool were the cost in terms of time spends using the tool, effectiveness of the tool in target coverage, and the quality of test cases generated from the Qtronic. This evaluation also includes some other aspects such as the usability and other higher level coverage facilities in the Qtronic.

Research Method

This thesis was conducted by two students working as test engineers in pairs. The period of this thesis was around 20 weeks. The starting two weeks were the first phase of research work for deep study of the MBT research papers, books and related documents. Then next two weeks were used to learn the Qtronic tool that includes reading the user manual and understanding the example model. An e-theater model was also constructed to practice tool. Based on the suggestion from the supervisor and the Conformiq assistance finally we selected our two case studies the Session Initiation Protocol (SIP) & File Transfer Protocol (FTP). For each case study, five weeks were used to read the protocol specification, construct the model, and generate both manual and automatic test cases. In each case study, we switched roles from automatic test engineer who used Qtronic tool to the manual test engineer. The purpose behind changing roles between engineers was:
• Their individual difference in experience and expertise of testing approach in generation of test cases.
• To generate separate automatic and manual test cases in order to compare.

The remaining five weeks were mainly used to construct, review and refine the thesis.

Efforts and Contributions

Efforts in this thesis include:

• Detailed Study of MBT books, research papers and relevant documents to get deep knowledge.
• Study of Qtronic tool manuals to get familiar with it.
• Modeling of SIP and FTP with modeler with use Qtronic Modeling Language (QML).
• Generation of test cases both with Qtronic and manually.
• Comparison of generation technique and test case quality.
• Experiments on Qtronic different coverage levels.
• Documentation of findings and results.

The main contributions of this thesis are:

• Evaluation of MBT methodology in general.
• A detail evaluation of Qtronic test case generation technique.
• Evaluation of Qtronic “lookahead” function.
• Evaluation of Qtronic test cases quality in comparison with manual test cases.
• Evaluation of general usability factors of Qtronic and modeler tool.
Chapter 1: Model Based Testing

1.1 Introduction of Model Based Testing

1.1.1 Background

“Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.” (Alain Abran, 2004).

A traditional and most widely used method of software testing is manual testing by using hand-crafted test suites. The manual testing is widely applied in most of software industries because of its simplicity, and ease to follow. Especially some cases are impossible to test only with test automation.

The product complexity is rapidly increasing while the requirement for the release time becoming shorter. The increased complexity of the product usually means that there will be possibly an infinite number of combinations of inputs and it is difficult for the traditional methods to cover most of them. Another problem is that the traditional manual testing stage is usually one of the last steps in the product development life cycle. The critical bugs found during the testing at later stages can cost time to fix and as a result the product release deadlines can be missed. (Mark Blackburn, 2004)

A lot of commercial automatic testing tools are used by the companies to reduce the time of the testing. Most of these tools are based on capture/playback mechanism. The tools can remember the action of the testers such as the mouse click or keyboard input. Such tools have some limitations for instance, they are fragile and if there is even a small change in the GUI, it can cause the failure of the test session. Furthermore, they may also require manual complementation and they don’t have requirement coverage functionality. (Mark Blackburn, 2004)
To minimize the limitations of current testing approaches, there was need of such methods that can generate test suits automatically at early stage of software development, and provide high coverage with less effort of work. In the mid-1970s MBT concepts were introduced. A brief description of the MBT will be described in upcoming sections.

1.1.2 What is model based testing

Mark and Bruno defined the MBT as

“Model-based testing is the automation of the design of black-box tests”. (Mark Utting B. L., 2007)

“black-box tests” explains the main scope of this methodology. The MBT is usually used in functional testing and it does not require understanding the internal code of the program. “Automation of the design” is the main difference between the MBT and other testing methods. The MBT is a testing method which can design the test cases automatically from the specifications.

1.1.3 The process of model based testing

A lot of research papers illustrated the process of the MBT. Following description of the MBT process is based on some chosen research papers. (Mark, 2005)(Mark Utting A. P., 2006)(Boberg, 2008)

1. The MBT process starts by constructing the abstract behavior models of the SUT and validating the model. The model is a representation of the intended behavior of the SUT based on the requirements and the test plan. After the construction of the models, validation process is applied to find the errors in the model.

2. The second step is generation of abstract test cases from the model. At the beginning of this step, it is required to define the test selection criterion. Some MBT tools also supports requirements traceability matrix which is used to check the mapping between the requirements and test cases.
3. The third step is transforming the abstract test cases into executable test cases or test scripts. The result of this step is a suite of test scripts which is applicable for executing the real testing.

4. The fourth step is execution of the test cases. Some MBT tools also provide the function of automatic testing based on the test scripts which usually need to write an adaptor between the MBT tools and SUT. The manual testing is another choice which does not require much programming skills.

5. The fifth step is analysis of test cases execution. Actual outputs are compared with the expected outputs and errors are detected in the product.

The figure 1 is the picture of the general MBT process life cycle described in above steps.
Figure 1: The MBT Process (Mark Utting B. L., 2007)
1.1.4 Benefits and Limitations

The MBT is a relatively new area. Industries are adopting this method because of its benefits during the development life cycle of product. Here is a list of some advantages of the MBT: (Mark Utting B. L., 2007), (Mark Blackburn, 2004).

- **More Fault Detection:**
  The results of some case studies, in which MBT implemented on some projects shows that more defects can be found by using the MBT. Two projects in the IBM implemented MBT. Results of their comparison with the manual testing show two additional defects were found in one project and four for the other project (E. Farchi, 2002). The results of another case study in Microsoft show that using MBT reveals ten times more number of defects. (M. Veanes)

- **Reduced Testing Cost and Time:**
  Under the help of the MBT tools, test cases generation and execution time can be reduced significantly. The requirements change only requires change in the model and that helps in saving a lot of time as compare to the manual design of the test cases. One result of a case study shows that nearly 90% cost saved after the use of MBT (Clarke J. M., 1998).

- **Improved test quality:**
  In traditional manual testing, the test cases are designed manually which means the test cases are produced non-systematically and they are also very difficult to manage. The quality of the test cases relies much on the experience of the test engineers. Where, the MBT uses tools to generate the test cases and record them. The different test case selection criteria can be selected according to the test plan and the quality of the test cases can be measured by the coverage.

- **Requirements Defect Detection:**
  The first step for MBT is creation of behavioral model of the SUT from the requirements. The errors or ambiguous points encountered during the building of the model usually reflect the defects in the requirements. This is one of the most important benefits of the MBT methods that it can reveal the faults in requirements in early stages of the lifecycle.

- **Traceability:**
  Most of the MBT tools also provide the traceability from the test cases to the requirements. This function makes the detection of the source of the faults easier.
The test engineers can quickly find which part of the requirements causing the fault.

- **Easy changing and updating:**
  All models can use the same test driver schema to produce test script for the requirements captured in each model. When the function of system changes are evolved, only the logic of model is needed to be changed, while when the test environment changes the test engineer just modify the test driver schema.

Every coin has two sides. Besides of all above advantages, the MBT also has some limitations. (Mark Utting B. L., 2007) (Arilo C. Dias Neto, 2007) (Robinson, 2003)

- Traditional manual testing does not require much skill for the testers however; the MBT requires the knowledge about the modeling and programming. This is the big challenge for the MBT testers. With less experience of modeling, the testers might spend much more time in learning and creating the model. The MBT tools also require a bit of programming, the testers will meet the obstacle if they do not have programming knowledge.
- Right now most of the MBT tools are used only for functional testing. They do not support the non-functional testing such as performance testing, usability testing, and security testing so on.
- Because of the complication of modeling and generation technique in the MBT, this method might not prove effective without any testing experience. It is better that the testers have some previous experience regarding the automatic testing.
1.2 Dimensions of the Model Based Testing

The MBT provides different dimensions in testing process of the SUT and it is up to the test engineer’s selection, to decide which is more suitable and effective for the application that is under test.

Utting, Pretschner, and Legeard in their article “A Taxonomy of Model Based Testing” describe seven different dimensions of the MBT. The Following figure is a complete map of possible dimensions of the MBT.

Figure 2: MBT Seven Dimensions. (Mark Utting A. P., 2006)
The vertical arrow in the diagram represents the alternative options for each specific dimension. For instance in the model subject dimension there are two alternatives that test engineer can choose either the environmental or behavioral model of the SUT. In other words the vertical arrows represent the possible ‘A/B’ alternatives at the leaves.

The curved lines represent all possible options that can be choose at same time. For example some tools might be able to use one or more generation technologies.

In the following sections detail description of each dimension represented in figure 2 can be found.

1.2.1 Subject of the Model

The modeling is the first step in the MBT. While building abstract model two possible dimensions are the environment model or behavioral model of the SUT. Mostly behavioral model is used but it’s possible that in some cases both dimensions can be used at same time.

The model in MBT served two ways; first it represents the behavior of SUT in this way it served as oracle to identify whether the model represents the correct behavior of the SUT. Second the same model is also used as generation of test cases. The environment model represents the interaction of the SUT to its environment in which it will perform its behaviors (functionalities). The behavioral model of the SUT represents the inputs, outputs and functionalities.

1.2.1.1 Levels of Abstraction

The model used in the MBT are very abstract, very less information is added as compared to original detail of the application. It also depends on the testing requirements; only functionalites required to test are added in the model.

There are different levels of possible abstraction that can be used in construction of the model that are “Functional Abstraction, Data Abstraction, Communication Abstraction and Abstraction of Quality of Service”.
i. **Functional Level Abstraction**

This level of abstraction is most widely used in the MBT. While building the model some of functionalities are omitted in order to make the models more abstract. The logic behind omitting the functionalities is that either some them are very simple, or has no effect on the behavior of selected functionality.

ii. **Data Level Abstraction**

Similarly in the data level abstraction both input and output of the SUT are omitted in order to simplify the models. Data abstraction level is also called input and output abstraction. By reducing the inputs from the model also reduced the number of test cases that simplifies test suits. Data abstraction may weaken the power of oracle.

iii. **Communication Level Abstraction**

The Communication abstraction level is often used in protocol testing where, it’s possible to represents handshaking between different layers by a single signal. It is also possible that to ignore sum of the signals to make the model simple.

iv. **Quality of Service Abstraction**

Abstraction from Quality-of-Service principle is often used to abstract from concerns such as timing, security, memory consumption, etc.

1. **2.2 Model Redundancy**

The MBT can be applied in many different scenarios. The difference among these scenarios is in redundancy level of model. Sometimes one model is used for both test cases and code generation. Another case is that when the separate models are used for testing and code generation.

Following sections will illustrate the above two scenarios.

i. **Single Model**

The single model scenario is using “One shared model for test cases and code”. Some CASE tools support generation of code from the model of system and same model then used for the test case generation. The models used for code generation are detailed model, because model covers all the intended requirements of system.
The idea behind using single model may be to test code generator and test cases at the same time. Another reason may be to save extra time spent on building the separate models.

Since detailed models are not suitable for testing perspective, the single model approach is not much effective in this sense

ii. Separate Model

The Second scenario is most widely used, “using separate models for testing purposes”. Main idea behind using separate model is that separate models are built for testing and implementation of the actual system.

The system is implemented manually using traditional software development process, based on existing formal specifications of the system. The models used in building architecture or implementation of the system are detailed models to represent the complete behavior of the system.

While in other case test models are built separately based on test requirements. Test models contain less detail and are very abstract in comparison to detailed behavior models used for implementation.

1.2.3 Model Characteristics

The model characteristics are non-determinism in the model, timing issues, and to the continuous or event-discrete nature of the model.

The models build for testing and the SUT both can be nondeterministic or deterministic. In some cases non determinism in the model can be used to test the deterministic systems.

The timing issues in the model are important in the Real Time Systems, because critical nature and criticality of time in every event in these systems it’s very hard to test them. Applications of the MBT in Real Time Systems are current research topics (K. Berkenkötter, 2005).

Finally, the models can be discrete, continuous or a mixture of both (hybrid). So far the MBT focused on event-discrete systems. The continuous and hybrid models are
often used in embedded systems. Similarly to the real time systems testing continuous system using the MBT are current research topics (K. Berkenkötter, 2005).

1.2.4 Model Paradigm

The model paradigm dimension of the MBT explains about the modeling notations and paradigms used in describing the model. Many modeling notations are available that have been used for the modeling the behavior of the SUT. One of the most widely used notations in the MBT is the UML state diagrams.

In the following sections we will summarize all of those paradigms grouped by Utting, Pretschner and Legeard, adapted from van Lamsweerde (Lamsweerde, 2000).

i. State-Based:

The State-Based notations represent system as a collection of internal states with some operations that represents the change from the one state to other. No code is required to represent the operations instead some preconditions and post-conditions are used for operation in each state. Some of the examples of such notations are Z, B, VDM and JML.

ii. Transition-Based:

The “Transition Based notations” used to represent the transitions between major states of the modeled system. Graphically these notations represented as collection of nodes and arcs, which are very similar to the finite state machines (FSMs). Nodes used to represent the states of system and arcs are used to describe the actions or operations of the system. The transitions between the states are described by some textual or tabular notations.

Some of the most common examples of transition based notations include FSMs, state charts, labeled transition systems and I/O automata.

iii. History-Based:

The History-Based notations are used to model system behavioral history traces over the time. Many time notations like discrete continuous, linear or branching, points or intervals etc. can be used to represent system history.

The message sequence charts and related notations can be included in this category.
iv. **Functional:**

In the functional notations system functionalities are described as a collection of the mathematical notations. The mathematical functions used may be first-order only, to avoid complexity in the model. The algebraic specifications are not common in the MBT because they are more abstract and difficult to write.

v. **Operational:**

The operational notations are normally used to represent the system processes executing parallel. The Distributed system and communication protocols are described using the notations. The Petri net notations are examples of such notations.

vi. **Stochastic:**

The Probabilistic model of the events and input values of the system are modeled using this group of notations. One example of such notations is, the Markov chains which is used to model expected usage profiles, then the generated test cases from the model can exercise that usage profile.

vii. **Data-Flow:**

The Data-Flow notations are used to describe the data flow of the system. Various data flow diagrams are common examples of such notations, for instance Lustre and the block diagrams.

1.2.5 Test Selection Criteria

The test selection criteria are used to control the generation of test cases. There is not a best criterion possible in general but it depends upon the test engineer to choose test selection criteria. Test selection criteria differ from system to system as testing requirements vary. Following sections most commonly used criteria in MBT are described.

i. **Structural Model Coverage:**

With the structural model coverage criteria nodes, arcs of transitions, condition statements and pre / post notations in the model are included.
The use of different modeling notations provides different structural coverage criteria. For instance, if the model consists of the UML state chart diagrams common structural model coverage criteria are all states or all transitions between states. There is many more other structural model coverage criteria that could possibly be used based upon which type of notations used in the model.

\textit{ii. Requirements-Based Coverage:}

When informal requirements are explicitly associated with the model then requirements coverage could be achieved by covering those. For instance, by attaching requirements numbers with transitions of the UML state machine could provide requirements traceability function in the model.

\textit{iii. Ad-hoc Test case Specification:}

The ad-hoc type of coverage criteria depends on test specifications to control the test case generation. Beside the use of the model built for test case generation, test engineer uses test specifications as guide for required type of test case generation. For example in the model, some paths are important to test so only test cases related to those paths will be generated.

\textit{iv. Random and Stochastic:}

The random and stochastic type of criteria mostly used in environment models to determine the usage pattern of the SUT. Then only the test cases following the expected usage patterns of the SUT are generated.

\textit{v. Fault-Based Criteria:}

One of the most common fault-based criteria is the mutation coverage. In the mutation coverage, a mutation version of the original model is created with some intended injected faults to make it different from the original model. The generated test cases based on this coverage criterion are aimed to detect these differences between these two versions.

\textbf{1.2.6 Test Generation Technology}

The big advantage MBT have over other testing methodologies is its automation in test case generation from the behavioral or environmental models. To generate test
cases from the abstract models a variety techniques are used that includes dedicated
graph search, algorithms, model checking, symbolic execution, or deductive
theorem proving.

In dedicated graph search methodology node or arc coverage algorithms are
included. One example of dedicated graph search is the Chinese Postman algorithm
which covers each arc at least once (Kwan, 1962)

The model checking technology is used for verifying certain properties of system.
Idea used in the model checking is transferring test case specifications to the
reachable properties of system then model checker helps to verify which states in the
model is reached or not.

In the symbolic execution technology an (executable) model is executed with sets of
possible inputs (e.g. [10-99]). The set of inputs are represented as constraints in
execution of model. The symbolic execution process is guided by test specifications.

Finally the theorem proving often used to check the verification of formulas where
such formulas used as guard in the state based model. One variation of this technique
is by replacing the model checker with the theorem proving.

1.2.7 Online or Offline Testing

The last dimension of the MBT concerns with two major types of testing Online and
Offline testing. The online and offline testing describes the time for the generation of
test cases and time for the execution of those test cases.

With the online testing test cases are generated and executed dynamically during the
running sate of the system. The online testing is essential when the SUT is non-
deterministic because it is hard to know which path the system will chose during
execution. But the online testing also gives extra burden to the development of
adapters for interfacing the SUT with CASE tool that used for the online testing.

On the other hand offline testing means test cases are generated from the abstract
model of the system and executed manually or automatically. No need to execute the
system, only functional properties of the systems are modeled and test cases are
generated by using that model. The offline testing gives many advantages to testers;
test case execution can be managed with traditional testing way, which means that
fewer changes in the test process are required. The regression testing is possible with the offline testing. Another advantage that offline testing has is test case generation and execution can be performed over different machines with different environments and at different times.

1.3 Previous Research on MBT

In this section we will describe some previous researches and experiments conducted over the MBT methodology.

The cost of testing always remains major concerns for the projects, and with the MBT how this cost effectiveness can be achieved also was big question. In order to determine what advantage can be achieved with the MBT in comparison with manual testing a research was conducted by James (Clarke J. M., 1998). In his research he compared the manual testing process with the MBT. Two major case studies were conducted by him, in his first case study he generated test case manually from the specification and in second time he created manually from the behavioral model. In order to compare with the MBT, he used TESTMASTER tool to generated test cases automatically from the model. The results of his comparison show that the MBT have increased the productivity to 90%.

Another research conducted at the Microsoft about using finite state machines (FSM) and Abstract state machine language (AsML) in the MBT (Stobie, 2005). By applying the FSM in many projects they show that there is need of some other more flexible modeling notation. They have also applied another notation called Abstract state machine language (AsML) in their experiments which increased the ability of the MBT in finding the defects in earlier stages of software development life cycle including specification and design stages. They also proved that by applying the AsML and its associated test tool (AsML / T) high coverage can also be reached.

A very detailed comparison research about the manual testing and the MBT was conducted by Preschner, W. Prenninger, M. Baumgartner, and T. Stauner (A. Pretschner, 2005). The application used in this research was network controller for
the modern automotive infotainment systems. Researchers built models for this application and created seven different test suites based on the selection of generation method (manual or automatic) and artifacts (using models and explicit test case specification or not).

After running those seven suites, they got some interesting results, the conclusions about them are:

- The tests derived without using a model caused fewer failures as compare to the model-based tests. The number of detected programming errors was approximately equal, but the number of detected requirements errors was higher in the model based tests.

- Where the automatically generated test suites detected same number of failures as compared to the handcrafted model based test suites, with the same number of tests. An increase in the number of automatic test case resulted in 11% percent increase in additional defect detection. None of the test suites detected all errors. With the hand crafted model based tests suits resulted in providing higher model coverage and lower implementation coverage than the automatically generated ones.

With increase in use of the MBT technique in various felids, some researches on the use of MBT in graphical user interface (GUI) testing were also conducted. In a research conducted by Qing Xie, a frame work for testing GUI of the SUT was proposed (Xie, 2006). A detailed process of the frame is shown in figure3. The core part of this framework was GUI model that will be used for test cases and oracle generation etc. Qing Xie developed experimental platform for his framework and applied it on various student projects. He concluded in his experiments that the MBT is feasible and have potential in applying it in the GUI testing.
Other researches were conducted in the health care and smart card industries. Marlon and his group applied MBT in the healthcare system and found that the MBT can help in fulfilling the coverage of the test cases for complex healthcare systems however there are also some challenges for applying the MBT in this particular area such as the preparation of the large amount of the test data and the training of the test analysts (Marlon Vieira, 2008). Another case study of automated test generation from a formal model of a smart card application makes it possible to automatically produce both the test cases and the traceability matrix (F. Bouquet E. J., 2005). Also, the MBT have provided numerous benefits for the overall software life cycle.

Figure 3: GUI Testing Frame Work (Xie, 2006).
Chapter 2: Constructing The Model

2.1 Steps of modeling

The construction of an abstract model of the system behavior or system environment is an important step in the MBT. The MBT tools generate test cases automatically that are based on an input model. The following steps are description about the construction of model. (Mark Utting B. L., 2007)

- **Decide on a good level of abstraction:**

  In this step, it is decided that which aspects of the SUT will be included in the model and which are not. In most of the cases, the systems are very large and complicated but the test models are only used for test generation, therefore the test model does not need to have to reflect all behaviors of the system and should keep model simple. It is always a good idea to split the whole system to some small components or sub-systems and construct the model for those small parts.

- **Consider about the data, the operation and the communication among subsystems:**

  If designers have created the class diagram for SUT, it can be used as a reference in the test model construction but cannot be used as test models. The system design models can be too complicated for testing purpose. When it is decided which input parameter will be included in the model, it required only to consider those inputs which will change the behavior of the system and then it is needed to leave the other parameters out of model. The reason is that the more inputs will be included the more test cases will be generated, which will increase the effort and complexity of testing.
• **Decide which notation will be used:**

In this step it is decided that what kind of notations the tools support and what kind of notations are most suitable for the SUT. Some common used notations are stated-based notation, transition-based notation, history-based notation, and data flow notations. A short description for those popular notations can be found in the previous section 1.2.

• **Validate and verify the model:**

Validation means making sure that the created models represents the behavior of SUT where verifying is to evaluate that the model is built correctly. Most of the MBT tools support this activity with some model checking functions.

### 2.2 Unified Model Language

The UML is a widely used modeling language for modeling in most of companies. Bouquet list two main reasons of why the UML is so popular (F. Bouquet C. G., 2007).

• There are a lot of different kinds of UML diagrams with different representations. The static representation such as the class diagram is used to display the static data function and the structure. The dynamic representation such as the activity diagram is usually used to represent the behavior of the SUT.

• UML is the de-facto industrial standard which means that most of the software engineers supposed to have some training of the UML.

The figure 4 is an example of State Machine Diagram based on a very interesting and small example. Here is the requirement from the customer: (Fowler, 2003)

“I want to keep my valuables in a safe that’s hard to find. So to reveal the lock to the safe, I have to remove a strategic candle from its holder, but this will reveal
the lock only while the door is closed. Once I can see the lock, I can insert my key to open the safe. For extra safety, I make sure that I can open the safe only if I replace the candle first. If a thief neglects this precaution, I'll unleash a nasty monster to devour him.”

![State Machine Diagram](image)

**Figure 4:** State Machine Diagram (Fowler, 2003)

The figure includes the following parts:

i. **Initial/Final State:**

Initial/Final States are not states but have an arrow that points to the initial/Final state of the system.

ii. **States:**

The diagram includes three states of the safe that are wait, lock, and open.

iii. **Transitions:**

The transition indicates a movement from one state to another. The transition can also be labeled by three parts trigger-signature [guard]/activity. All these
parts are optional. The trigger-signature is usually a single event that triggers a potential change of the state. The guard, if present, is the Boolean condition that must be true for the transition to be taken. The activity is combinations of behaviors that are executed during the transition. It may be any behavioral expression. The full form of a trigger-signature may include multiple events and parameters. In above figure, the transition from wait to lock includes these three parts. This transition means, if the candle is removed (trigger) and the door is closed (guard), then the state of the safe is transfer from wait to lock and the lock is revealed (activity).
Chapter 3: Conformiq Qtronic™

3.1 The Conformiq Company

The Conformiq Company was founded in 1998 in Finland. The R&D facility of the Conformiq is located in the Finland where headquarters of the Conformiq are located in the Saratoga, California. Sales offices of the Conformiq are located in the United States, Finland and Sweden. (Conformiq, About Us, 2009)

3.2 Conformiq Qtronic™ Introduction

The Qtronic 2.0 is an Eclipse based tool released in 2008. Qtronic automates the design of functional test cases for the SUT. The Qtronic tool can derive the functional test cases from an abstract behavioral or environmental model of SUT or device under test (DUT). Qtronic 2.0 has only offline testing capability where online testing was included in previous versions. The Qtronic uses UML state chart diagrams for building the models, where it has its own textual java based modeling language QML (Qtronic modeling language). The Conformiq Company has provided extra tool for modeling purpose called the Qtronic Modeler but it is possible to use third party tool for the modeling. (Conformiq, About Us, 2009)

3.3 Automated Test Generation Flow with Qtronic

Conformiq describes the generation of automated test cases as a three step process. Detail of each of these steps is as follows. (Conformiq, Qtronic Brochure, 2009)
1. Model Creation:

In the first step an abstract model is constructed with combination of the UML state chart diagrams and QML. The Conformiq has provided a lightweight tool, the Conformiq Modeler for modeling but the third party UML modeling tool can be used for the creation of model. Constructed model then used as input to the Qtronic for generation of test cases.

2. Test Case Generation:

The Qtronic checks input model for correctness first and immediately reports error if some error exists in the model. On the success of model checking the Qtronic automatically generates executable test cases and associated test plan documentation, including traceability matrices for requirements and state transitions, message sequence charts, etc.

3. Test Execution:

The generated test cases can be exported as test scripts, the supported script formats are

i. HTML (Hyper Text Markup Language)

ii. TTCN-3 (Testing and Test Control Notation Version 3)

iii. TCN (Tool Command Language)

It is also possible to develop plug-ins by ourselves, or contract the Conformiq C2S2™ services to create plug-ins that generates desired output formats.

3.4 The Qtronic Coverage Criteria

The Qtronic Coverage Criteria are based on model driven coverage. The Qtronic tool provides different coverage criteria on the bases of model structure. Following are
the details of each coverage criterion available in Qtronic tool. (Conformiq, Qtronic2x Manual, 2009)

a) Requirements Coverage

In the requirements coverage every requirement in the model will become a test goal and only the test cases that covers the specific requirements are generated. By attaching requirements with every transition in the model can also help in tracing the requirements during the test case generation.

b) Transition Coverage

In the transition coverage every UML state chart transition in the model will become a test goal, and the Qtronic will generate test cases to cover the transitions in the model.

c) State Coverage

In state coverage criteria every state in the model considers as test goal and the Qtronic will generate test cases to cover the states of the model.

d) 2-Transition Coverage

In 2-transition coverage criteria every sequence of two transitions that can be executed in sequence with a single state machine becomes test goal. Only those test cases that will cover the sequence of two transitions will be generated.

e) Control Flow Coverage

In control flow coverage, the Qtronic tool considers test goal to every “then” and “else” branch of conditional (If statement) and also the bodies of every while and for loop.

f) Atomic Condition Coverage

In the atomic condition coverage every atomic Boolean expression (a && b), guides the Qtronic to look for behaviors that cover every QML level atomic condition branch such as left and right hand sides of a Boolean && (and) at least once.
g) **Boundary Value Analysis**

For the boundary value analysis, the Qtronic tool considers four testing goals for arithmetic inequality expression two around the decision boundary and two outside it. For equality and non-equality expressions one testing goal is at equality and two are on the both sides of non-equality (<, >, <=, >=) boundary.

h) **Statement Coverage**

In the statement coverage criteria every statement either in the model or in the text files becomes the test goal for the Qtronic.

i) **Method Coverage**

In the method coverage criteria, every method defined in the model becomes the test goal for the Qtronic and considered as coverage criteria.

j) **All Paths Coverage**

In all path coverage, every distinct sequence of path in the model becomes the coverage criteria for the Qtronic.

k) **Implicit Consumption**

“Implicit consumption in the UML means that a message that is not handled actively in the current state is discarded automatically”. If this setting is checked in the Qtronic it allows the test cases that result in implicit consumption.

### 3.5 Lookahead Depth

“Controls the amount of lookahead for planning the test scripts. The value of the lookahead corresponds to the number of external input events to the system or timeouts. When Qtronic plans the tests, it intellectually selects interesting values for data based on the logic in the design model. If the logic that manipulates the data is after certain number of external events, the lookahead value must be increased, because Qtronic must be able to "see" this in order to make decisions on the data values”. (Conformiq, Qtronic2x Manual, 2009)
The purpose of this chapter is to explain the Qtronic Modeling Language (QML). In order to explain the QML a simple example is used to show the general picture. The example is from the Conformiq and it is representation of the SIP. The models and the QML code can be found in Appendix A.

The model is very similar to the example described in the previous chapters Section 2. Both of them include states, transition and notes. The main difference between these models is that in the previous example natural language description was used as representation of the requirements in the transitions while action code is used in the QML.

As described in the chapter 3 the action code in Qtronic is java based language called QML. Considering the purpose of this thesis, our aim is not to describe the use of this tool but we will provide an overview of the QML structure. The Conformiq has provided a detail description of the QML in user manual. (Conformiq, Qtronic2x Manual, 2009)

4.1 System Block

The first part of the QML is the system block which is definition of the environment of SUT including the inbound and outbound port and messages:

```qml
system
{
    Inbound userIn  : UserInput;
}
Inbound netIn : SIPResp, SIPReq;
Outbound netOut : SIPResp, SIPReq;
Outbound userOut : TimeOutIndication;
}

“userIn” is the name of the port and “UserInput” is the name of the message.

4.2 Record

After defining the system, each message should also be defined. In the QML, the message is called record. The definition of the record is very similar to the definition of the class in Java or Struct in the C++:

record UserInput
{
  public String input1;
  public String input2;
}

4.3 Methods

The next step in QML is creation of the methods which are used in the guard or action part in the transition of model. The method declaration and definition is very similar to the Java:

public void Invite()
{
  SIPReq r;
  r.op = "INVITE";
  r.param = dst;
  netOut.send(r, 1.0);
}

The method should be defined in the class which is defined as:

class SIPClient extends StateMachine {}

Here the “extends StateMachine” used to represent that this class is combined with the model which has the same name as the class name.
4.4 Main Method

The final part is the main method for running the QML code which is defined as this way.

```cpp
void main()
{
    var a = new SIPClient();
    a.start();
}
```

4.5 Key Words

Besides the general part, the Qtronic tool also included some special keywords or statements in the SIP example.

1. The keyword “requirement” in the model represents the requirements for the SUT and it is also used in requirements coverage matrix for tracing the coverage.
2. The keyword “after” in the model is a time limit method. It is usually used as a trigger in the transition.
3. The keyword “require” in the model or action code is the keyword which means that the condition after this keyword should be true.
4. The format of the trigger in the transition is “port:message”. For example, “userIn:UserInput”.
5. The keyword “msg” in the guard of the transition represents the record used in this transition.
Chapter 5: Case Studies

This chapter describes our practical case studies, experiments and the results. Mainly two case studies were conducted entitled SIP and FTP.

Pre-Study

In order to get familiar with the Qtronic tool, a web application called e-theater was modeled for test case generation. It was experienced that with basic knowledge of the java and state machine diagrams getting familiar with the Qtronic proved easier. It was also observed that the Qtronic tool is not much suitable for web based and GUI testing because the Qtronic was specially designed for communication domain. As “msg” keyword and time out functions shows intention of Qtronic tool towards communication sector.

The first case study, SIP was proposed by Conformiq by providing an example model, specification and continuous guidance throughout our practical work. This proved very helpful. The second case study (FTP) was selected to experiment some specific dimensions of the MBT by using Qtronic. A detailed description of each of the case studies and their results will be illustrated in upcoming sections.

Quality of test suites

The test suites are a set of test cases. As the test suites are used to test the SUT, the direct way to measure the quality of test suites is to run them and measure the number of the detected fault and the execution time. A good quality test suite should reveal more faults in a short time.
However, the Qtronic and manual test suites from the case studies are not intended to be executed therefore the quality cannot be measured based on the number of faults. Instead both the Qtronic and manual test suites were generated based on the same coverage criteria.

In the SIP case study requirements coverage, all state coverage and all transition coverage were selected. In the FTP case study all state, all transitions and atomic coverage were included. Because both the Qtronic and manual test suites are generated with same coverage criteria, they should generate very similar number of fault therefore with the short time consumption, less number of test cases or test steps reflects the less time of execution and the better quality in some extend. Besides more coverage criteria are also selected in Qtronic to see how many extra test cases are generated. More coverage criteria usually reveal more fault detection, if the Qtronic can apply those extra coverage criteria in acceptable time and can generate more test cases, we can say that Qtronic has ability to generate better quality test suites.

**Representation of Test Case**

The test cases generated from the Qtronic are abstract test cases. They only include necessary information for executing the real testing. The tester can transfer these abstract test cases to executable test scripts. If the tester has enough experience, they might not require transferring test cases and can execute the real testing directly from these abstract test cases.

In the Qtronic the test cases can be represented in 2 ways. The figure6 and figure7 are two examples which display the interaction between tester and SUT or a description of the sequence of the test steps.
Figure 6: Qtronic Test Case Tester Interaction

Figure 7: Qtronic Test Case Steps
In these two case studies the manual test cases were also generated. Because, the purpose of this thesis was compare both automatic and manual test case generation methods without executing them.

The format of manual test cases is similar to the test cases from Qtronic and these manual test cases are also abstract. The table 1 is an example of manual test case.

<table>
<thead>
<tr>
<th>Timer</th>
<th>Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>The TU input “INVITE” request to SUT.</td>
</tr>
<tr>
<td>0.0</td>
<td>2</td>
<td>The SUT passes the “INVITE” request to the transport layer for transmission.</td>
</tr>
<tr>
<td>0.0</td>
<td>3</td>
<td>The transport layer input status code “149” response to SUT.</td>
</tr>
<tr>
<td>0.0</td>
<td>4</td>
<td>The SUT passes “149” response to the transaction user.</td>
</tr>
<tr>
<td>0.0</td>
<td>5</td>
<td>The transport layer input status code “332” response to SUT.</td>
</tr>
<tr>
<td>0.0</td>
<td>6</td>
<td>The SUT generates an ACK request and passes it to the transport layer for transmission.</td>
</tr>
<tr>
<td>0.0</td>
<td>7</td>
<td>The SUT passes this response to the transaction user.</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>The SUT send timer D time out indication to TU.</td>
</tr>
</tbody>
</table>

Table 1: Manual Test Case
5.1 Case Study 1: SIP

5.1.1 Introduction

This case study was about the SIP. The transactions part of the SIP protocol was modeled for the test case generation purpose. The created models have strictly followed the requirements. (RFCeditor, 2009)

The process and requirements for the transaction part is in the section 17 of the protocol. There are four kinds of transactions:

- Invite client transactions.
- Non-invite client transactions.
- Invite server transactions.
- Non-invite server transactions.

A detailed description of each model and QML code can be found in Appendix B.

5.1.2 Method

In this case study we have followed the dimensions of MBT that we have described in Chapter 1: behavioral model, separate test model, deterministic, with state machine notations, test case selection criteria was structural model coverage and requirements coverage. The technologies used for test case generation were manual and automatic by using Qtronic. The last dimension was out of the scope of our thesis, which was method of test execution because we didn’t execute the test cases.

Our main method was started from creating the models according to the SIP specifications. Then using the same model, one tester generated test cases manually from the model and another tester used the Qtronic tool to generate test cases from the same model. Then we compared both manual and automatic test cases sets. Since the test selection criteria were requirement and transition coverage. It means that the test cases should include all requirements and transitions in the model.
Besides the requirement and transition coverage, we had also experimented some other coverage criteria such as “2-transition”, “Boundary value” etc. The results of those experiments are also included in the result section of this case study.

5.1.3 Case Study 1 Results

The following section describes the results of SIP case study that we got after comparing manually produced test cases and automatically generated test cases produced by Qtronic. We compared, time consumption, number of test cases, test case steps of both methods and we also analyzed some other aspects like higher coverage criteria.

Here are our findings about this case study.

a) Manual test case generation took 4.5 hours and validation of these test cases took 1.5 hours, while 0.5 hours was used to fix calculation problems found in some manual test cases. Whereas in the case of Qtronic tool, the first three models (invite client model, non-invite model and invite server model) took 0.5 hour each only in test case generation. The reason was that we increased the “lookahead” depth level to the third level in order to reach the timeout requirements therefore increasing the computation time. We used the default “lookahead depth” level (the lowest) on the last model (non-invite server) and it only took 2 seconds. The Qtronic tool took 1.5 hours in total for generating the test cases which was 1/4 of the time spent on manual test case generation.

![Figure 8: Time Comparison](image)

Hours

<table>
<thead>
<tr>
<th></th>
<th>Automatic</th>
<th>Manual Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

36
b) In manual test case design, totally 33 test cases were created for fulfilling the requirements and transitions converge. With the same criteria, the Qtronic tool generated the exactly same number of test cases for each model.

<table>
<thead>
<tr>
<th>Models</th>
<th>Qtronic Test Cases</th>
<th>Manual Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invite Client</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Non-Invite Client</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Invite Server</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Non-Invite Server</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>TOTAL</td>
<td>33</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 2: Number of Test Cases for SIP

c) Manual testing is good when the customers only require requirements level coverage and the model is not too complex. In this case study, the test cases generated manually are very similar to the test cases generated by the Qtronic but both of them only consider the basic requirements and transitions converge. If the customer needs to include more criteria such as the boundary value, atomic value (true, false situation) and so on it will definitely spend much more time and introduce more logic problems for manual test case generator. On the other hand, the Qtronic can do this job in one click, just required to change the coverage criteria, and the Qtronic will generate them automatically. The extra time spent can be acceptable and rarely includes logical problem if the model itself is correct.

d) We also tried other coverage levels by changing coverage criteria for all models the Qtronic works for only two models in the acceptable time. The other two models failed because of the long progress time taken by the Qtronic. In the invite client model, we included the boundary value and atomic value testing the 4 extra test cases were added by Qtronic tool. In the non invite server model, we included all criteria such as the boundary value, atomic value, control flow, two
transition and implicit consumption. The result was that Qtronic added 31 test cases with only 2 seconds extra time.

e) When the tester created the manual test cases in this case study, he focused more on the exchange of messages which is the most important testing purpose. To keep the work simple and time controllable, the tester ignored some other details. On the other hand, when we checked the test cases from the Qtronic, includes more information than the manual one. The most obvious point is about the SIP message itself. The test cases in Qtronic give the whole contents of the message which includes method name, URI and headers. This will reduce the future work when we execute these test scripts.

f) In manual test cases, most of steps are repeated in every test case because it’s hard for manual test engineer to keep track of the transitions that are covered and uncovered. This problem resulted in testing the timeout function repeatedly as this function spends more than 32 seconds in execution every time that might cause time consuming for tester at the end. On the other hand Qtronic covers the timeout function only once and avoids the repetition.

g) During the manual test cases generation, the tester usually first decided a main stream in the model and treated other states as the leaves e.g. “state 6” in figure 9. Although it was comprehensive it sometimes resulted in increasing the complexity and steps of the test case which used to test those leaves. In case of, the Qtronic tool usually select the shortest path.

The following figure explains the comparison.
The manual tester usually first selected state “1-2-3-4-5” as main stream and treated state 6 as leaves due to the layout of the figure, when he wrote the test case to test the transition between state 4 and 6, resulting test case is like “1-2-3-4-6”. Where path selected by Qtronic tool includes “1-4-6” states instead.

h) The numbers of the total test steps are reflected in the comparison points d and e. We showed that the number of test cases for both cases was exactly the same. But the result of the total test case steps of the manual one was 250 and 213 for the Qtronic tool.
5.1.4 Further investigation about “Lookahead” depth

In this section, we further discussed about the “Lookahead” depth. Since the last section, it was mentioned that the first three models (Invite Client, Invite Server, and Non-invite Client) need increase in the “Lookahead” depth to the third level to fulfill all requirements. Also, the SIP example provided by Conformiq has the same need. But, at the same time, the last model (Non-invite Server) does not need to increase the level. In this short section, we tried to find out the reason behind this.

By comparing the Non-Invite Server model with other 4 SIP models, the difference was that no sub-states were included in the Non-Invite Server model. Another interesting finding was that for those 4 complex models, all unreached requirements under the lowest “Lookahead“level were about timeout. So we did some further experiments to investigate these points.

The following tables are the summary of our investigation. Four experiments were conducted. We created a table for each experiment. Each table includes the motivation of the experiment, example model, the lowest lookahead level for 100% coverage and the conclusion of experiment.
Experiment 1:
We simplified the main model to a smaller one without changing the sub model.

Result
The complexity of the main model does not affect the selection of the level.

Table 3: "Lookahead" Experiment 1

Experiment 2:
Eliminating both loop and the time out condition from sub model.

Result
The factor may be the loop or the time out or may be both.

Table 4: "Lookahead" Experiment 2
**Experiment 3:**
Including loop conditions only.

#### Table 5: "Lookahead" Experiment 3

**Result**
Loop is not causing factor only.

---

**Experiment 4:**
Adding timeout condition only.

#### Table 6: "Lookahead" Experiment 4

**Result**
Time out condition is not only one effecting factor.

---

**Final Conclusion:**

Finlay we can conclude that there is not only one factor that causes the change in "lookahead" level most possible factors to cause the increase in level are the combination of loop and time out condition in the sub model.
5.2 Case Study 2: FTP

5.2.1 Introduction

This case study was about FTP. The aim in this case was to use such models that have more conditions, decisions and transitions, to evaluate Qtronic. We used FTP requirements specifications to model the command sequence part of FTP. Created models had strictly followed the specifications of FTP. A detailed description about the FTP and FTP specification used in this case study can be found Internet Spec List site. (Productions, 2009)

Five models were created for the FTP case study. The models consisted of only selected FTP commands. Our first model was about the “ABOR, HELP, and MODE” commands, second model consisted of the “LiST, RETR, and STOR” commands, third model was only for the rename sequence “RNFR, RNTO” commands, fourth model was for the restart command “REST” and the fifth model the most complicated was for the login sequence “USER, PASS, ACCT” commands. The detailed description of each model can be found in appendix C.

5.2.2 Method

In this case study, the dimensions of MBT were similar to our first case study. Dimensions we have followed were, the model we constructed was behavioral model, separate test model, deterministic, with the state machine notations, test selection criteria was structural model coverage and requirements coverage. The technologies used for the test case generation were manual and the technology used by Qtronic for automatic test case generation. The last dimension was the same again, out of the scope of our thesis that was method of test execution.

In this case we changed the method of manual test case generation, we generated the automatic test cases using the Qtronic from the models and manual test cases were generated from the requirements specifications. In the FTP specifications document state diagrams are also available to give clear understanding of the command sequence. Our method is changed from our previous case study in which we generated both manual and automatic test cases from the same models. Again we
have compared the manual and automatically generated test cases to observe the differences.

5.2.3 Results

In this section we will illustrate the results of our FTP case study. Similar to previous case study there was one test engineer involved in automatic test case generation using the Qtronic and one test engineer in manual test case generation. The comparison between manual and automatic produced test cases is as follows.

a) The number of test cases produced by the Qtronic tool was 107, while the test cases produced manually from requirements specifications were 132. Number of the test cases produced for each set of the FTP command sequence are shown in table 6.

<table>
<thead>
<tr>
<th>Command Sequence</th>
<th>Qtronic Test Cases</th>
<th>Manual Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABOR, HELP, MODE</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>LIST, RETR, STOR</td>
<td>22</td>
<td>40</td>
</tr>
<tr>
<td>RNFR, RNTO</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>REST, APPE</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>USER, PASS, ACCT</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>107</strong></td>
<td><strong>132</strong></td>
</tr>
</tbody>
</table>

Table 7: Number of Test Cases for FTP

b) Why the Qtronic did produce fewer test cases than manual? In the first and second sets of command sequences the Qtronic have almost half of test cases than manually produced test cases. In order to answer this question we compared each automatic test case with manual test cases and we found interesting results. In the first and second set of command sequences have lot of condition points in the model such as (A || B) means A or B. To cover every A or B conditions there should be a lot of combination of test cases but the Qtronic tool only covered one
combination of each condition, where manual tester generated test cases for each combination. Here is a simple example to give a more clear explanation.

**Figure 10:** Example for the atomic condition selection

If we consider figure 10 as an example, then manual test cases for each combination will be \((3 \times 2 + 3 \times 2 = 12)\) 12 test cases but the Qtronic tool produced only 4 test cases (af, bg, cd, ce) covering only each condition.

So that was reason of fewer test cases produced by Qtronic. In Qtronic Manual this was mentioned that Qtronic cover every atomic condition branch at least once (Conformiq, Qtronic2x Manual, 2009).

c) In third set of commands there was only one test case difference between manual and Qtronic test cases. The reason for this difference was a slight mistake of manual tester; he missed one command code from specification.

d) For the fourth and fifth sets of commands manual tester have 2 test cases less than the Qtronic test cases. After comparing the test cases we found the answer. The next figure will explain that reason.
Considering figure 11 the Qtronic tool will produce two test cases one for transition 1 and one for transition 2. The manual tester produced one test case covering both transitions in one test case. That was the reason of Qtronic generated two extra test cases for the fourth and fifth command sequence.

e) We also compared the test case steps and quality of each step in both Qtronic and manually generated test cases. The number of steps in test cases in both cases was equal only in last 2 models of FTP test steps of some test cases were different because of the reason explained in section (d).

f) The time taken in producing manual test case was equal to time taken by automatic tester in modeling. After modeling, the Qtronic only took 1 second for generating the test cases. The reason was due to of simplicity of the requirements and had no extra calculation burden like timeout calculation in the SIP case.
5.3 Reflection of Manual Test Case Generation

This section will illustrate the experience of manual test engineer in generating manual test cases from the models and specifications that we developed in our case studies.

Here is a description of manual test case generator reflection.

- For some calculations like timer calculation in the SIP case study, manual test engineer has to calculate the timer of each message manually and needed to describe the timer with every message which is time consuming and error prone.

- It was very hard for the manual tester to keep track of transitions and path covered by manual test cases. The manual tester had to mark each covered requirements in the model that slow down his speed.

- Considering only requirements coverage in the manual test cases generation, it was lot easier than full coverage. Because of the time limitation manual tester was only considering requirements and transitions level coverage. But it is very difficult to achieve higher cover like the boundary value, full path coverage etc. manually.

- Since manual test engineer created hand crafted test cases, it was possible that some typing mistakes or some wrong step calculation may occurred that might have lead to a wrong test case in the end.

- As there was no sophisticated methods to keep track which transition is already covered or not, there were possibly repetitions of steps in some test cases that could have increased the time of testing when these test cases will be used.

5.4 Reflection of Automatic Test Case Generation

This section includes the description about Automatic test engineer experiences in generating automatic test cases using Qtronic tool from the SIP and FTP models.
We can generate test case from the Qtronic tool with one click but still it requires some time in writing QML code for the model of inbound and outbound functions.

Following is the reflection of automatic test engineer that he experienced during these case studies.

- The test case generation process was totally automatic this task can be run during the leisure time or night, this was very convenient for the tester and the results can be checked in the next morning or after the break. Where such flexibility can also increases the effort of the project work on other tasks.

- The model checking function was very helpful for checking the errors in the model and QML code before creating test cases. After modeling, the tester can select "load model" function in Qtronic to do model checking. The Qtronic tool will check both state machine diagram and QML code. The logical faults in diagram or errors in code will be displayed in a console window. If the fault is in the diagram, the console will show which transition included in this fault. If there are errors in QML code, the number of the line will be displayed with some hints or suggestions.

- With the Qtronic tool it is very easy to manage the requirement change problem; sometimes it was only required to change the model to fulfill new requirements even without changing QML code. That saved our lot of time in case studies.

- However, modeler tool is not much intelligent in transition placement. In some of our models, there are lots of transitions and sometimes we need to adjust the position of the transition line, but it is not easy especially if the transition includes very long text (triggers, guards, actions and requirements). Another thing, when we were drawing one transition between two states the transition cannot be connected automatically by the modeler tool. If we draw it carelessly, the transition will not connect to the states and an error will happen during the model checking process.

- There was no definition or principle of the different level of "Lookahead" depth and the default level was too low. There are seven "Lookahead" levels in Qtronic,
and the default level is the lowest level. Since we used four models in this case study, at first only using the default level of “Lookahead” to generate the test cases with requirement and state coverage criteria. In three models the default level cannot fulfill all requirements and we have to increase it. Considering that the models in the SIP case study are not very complex we therefore think the default level should be on another higher level. In the user manual, the Conformiq suggests that in most of the cases, the third level should be the highest level because if a higher level is selected it will spend much more time and if still some requirements cannot be reached in the third level it usually means that there are some problems in the model. Considering this suggestion, we think it is better that the first three levels which are used often can be divided into more levels and the last four levels which are rarely used can be compressed to only one or two levels. Also, if there are some principles or suggestions for selecting the correct level, it will save the time for trying each level from the lowest one. (Conformiq, Qtronic2x Manual, 2009)

- During the test case generation process with Qtronic, a status bar in the console window of eclipse shows the percentage of the progress. In the SIP case study, after fulfilling the requirements and the state coverage for all models we tried to include “2-transition” and “implicit consumption” coverage. During that task in the third level, the Qtronic tool started the job but the progress in the status bar was always 0%. We expect that this job will take a lot of time, but after 1 hour, the progress was still 0% and then we had to stop this task.
Conclusion & Future Work

This section includes brief description of the conclusion that we have derived by analyzing the results of our case studies, reflection of test engineers, experiments and from the theory. We also stated what needs to be done in the future in order to conduct a comprehensive evaluation of the MBT and Qtronic tool.

The MBT method provides a way that involved testing in the early stages of the software development process, because requirements specifications can be directly modeled. Modeling the behavior model of the SUT from the requirement specifications, the defects and ambiguous points in the specification can be found and corrected at very early stages. This can reduce a lot of effort and time because the defects found in the earlier stages costs much less than the later one.

With automatic test case generation facility, the MBT keeps greater advantages even the generated test cases are very abstract and cannot be executed directly. For instance, with automatic test cases some mathematical mistakes, wrong calculations and test coverage problems can be easily avoidable as most of the CASE tools like the Qtronic have such basic built-in facilities.

The change of requirements is one of the big and painstaking problems in software development life cycle which can be easily handled in the MBT. With change in requirements only required slight changes in the models and with one more click new test suits are generated in short time.

The MBT is not only facilitated with automatic generation of test cases it also have some other basic facilities like criteria selection, model checking and test case traceability matrix.

Since, Qtronic tool plugged in to the Eclipse IDE and the QML similarity with Java language makes Qtronic very convenient for test engineers that require only basic knowledge about java.

The Qtronic tool is targeted especially for the communication domain applications, the timeout function and msg keywords are designed especially for such areas. That might was reason that the Qtronic is not much suitable for GUI and Web testing.
The modeler tool provided with Qtronic requires some improvements. Especially messages with transactions are hard to handle and difficulty to find which transaction arrow is not properly attached to the state.

Like the other MBT tools, The Qtronic provides a lot of convenient functions like model checking function, different coverage selection criterion, requirements traceability matrix which proves very helpful and effective in test generation. Also generated test cases can be exported in three different scripting formats, use of HTML format proved very effective in test case comparison of our case studies.

Two case studies SIP and FTP conducted in this thesis shows effectiveness of the Qtronic tool in comparison with the traditional manual test case generation method. The SIP case study results shows that test case generation is quick and less time consuming than manual generation because of timeout conditions and message passing involved in models which are easy to handle with the Qtronic but hard with manual process. It also shows the Qtronic efficiency of avoiding repetition of test steps, and providing power of higher coverage criteria. However, the interesting “Lookahead” function produces some problems in our cases studies such as the computer freezing and time consumption. We expect that it will be improved with a more clear division level structure and better instructions about how to choose a suitable “Lookhead” level.

However our FTP case study results shows that when the models are simple without any timeliness messages and with simple functional behavior, the efficiency of automatic and manual generation process proved almost equal. In such cases either of both test generation techniques can be more efficient.

With limitations of the time and resources the scope of this thesis was only limited to the offline test case generation. For future work we would like to execute generated test cases in order to analyze quality and effectiveness of test cases. We are also would like to perform test cases execution. In future, with better resources and time we would like to expand the scope of our research work in order to cover other dimensions of the MBT.
References


Conformiq. (04 2009). *About Us*. Hämtat från Automated Test Design:

Conformiq. (April 2009). *Qtronic Brochure*. Hämtat från Automated Test Desing:
http://www.conformiq.com/resources.php den 28 April 2009

Conformiq. (April 2009). *Qtronic2x Manual*. Hämtat från Automated Test Desing:
http://www.conformiq.com/resources.php den 28 April 2009


Appendix A: SIP Example From Conformiq

Main Model
Appendix A

Terminating Model

Calling Model
Code

// -*- cqa -*-

/* System Description
 * Defines input/output ports,
 * and valid messages per port.
 */
system
{
    Inbound userIn : UserInput;
    Inbound netIn   : SIPResp, SIPReq;
    Outbound netOut : SIPResp, SIPReq;
    Outbound userOut : TimeOutIndication;
}

/* ====================================================
 * Message Definitions
 */
record TimeOutIndication { }
record UserInput
{
    public String input1;
    public String input2;
}
record SIPResp
public int status;
public String cseq;
}

record SIPReq {
    public String op;
    public String param;
}

/* ====================================================
 * SIPClient
 */
class SIPClient extends StateMachine {
    const float T1 = 0.5; // For UDP transport

    public float timeoutA = T1; // 17.1.1.2
    public float timeoutB = 16 * T1; // 17.1.1.2
    public float timeoutE = T1; // 17.1.2.2
    public float timeoutF = 16 * T1; // 17.1.2.2

    /* -----------------------------------------
     * SIP addresses
     * src = caller, dst = callee
     */
    public String src = "sip:127.0.0.1";
    public String dst = "sip:127.0.0.1:5061";

    /* -----------------------------------------
     * Methods to send SIP messages to network.
     */
    public void Invite() {
        SIPReq r;
        r.op = "INVITE";
        r.param = dst;
        netOut.send(r, 1.0);
    }

    protected void Cancel() {
        SIPReq r;
        r.op = "CANCEL";
        r.param = dst;
        netOut.send(r, 1.0);
    }

    protected void Ack() {
        SIPReq r;
        r.op = "ACK";
        r.param = dst;
        netOut.send(r, 1.0);
    }

    protected void Bye() {
        SIPReq r;
Appendix A

```java
    r.op = "BYE";
    r.param = dst;
    netOut.send(r, 1.0);
}

protected void SendOK(String cseq) {
    SIPResp r;
    r.status = 200;
    r.cseq = cseq;
    netOut.send(r, 1.0);
}

/**
   * Methods to provide indications to the
   * user interface.
   */
private void TimeOut()
{
    // Indicate timeout
    TimeOutIndication timeout;
    userOut.send(timeout);
}

/**
   * main() - Starting Point
   */
void main()
{
    var a = new SIPClient();
    a.start();
}
Appendix B: SIP Case Study Model

B1: Invite Client Transaction Model

Evaluation of Model Based Testing and Conforming Qronic
class INVITEClientTransaction extends StateMachine
{
    public INVITEClientTransaction()
    {
        timerA = T1;    // == 0.5
        timerB = 64*T1; // == 32
        timerD = 32;
    }

    protected void sendInvite(SIPRequest invite)
    {
        require invite.method == "INVITE";
        require invite.requestURI == "sip:bob@wonderland.com";
        requireHeaders(invite);
        originalRequest = invite;
        transportOut.send(invite);
    }

    protected void sendTimeout(String timerName)
    {
        Timeout t;
        t.timerName = timerName;
        tuOut.send(t);
    }

    protected void forwardResponseToTu(SIPResponse resp)
    {
        requireHeaders(resp, originalRequest);
        tuOut.send(resp);
    }

    protected void forwardErrorToTu(TransportError er)
    {
        tuOut.send(er);
    }
protected void sendAck(SIPRequest invite, SIPResponse ackTo) {
    SIPRequest ack;
    requirement "17.1.1.3/1q. ACK request constructed by the client transaction MUST contain values for the Call-ID, From, and Request-URI that are equal to the values of those header fields in the original request.";
    requirement "17.1.1.3/1s. ACK request must contain a single Via header field and this must be equal to the top Via header of the original request.";
    requirement "17.1.1.3/1r. The To header field in the ACK must equal the To header field in the response being acknowledged.";
    requirement "17.1.1.3/1t. CSeq header field in the ACK must contain the same value for the sequence number as was present in the original request but the method parameter must be equal to 'ACK'.";
    ack = invite;
    ack.headers.cseq.method = "ACK";
    ack.headers.to = ackTo.headers.to;
    transportOut.send(ack);
}

protected boolean statusInRange(int statusCode, int lowerBound, int upperBound) {
    return statusCode >= lowerBound && statusCode <= upperBound;
}

private void requireHeaders(SIPRequest invite) {
    require invite.headers.callID == "1";
    require invite.headers.cseq.seqNo == "1";
    require invite.headers.cseq.method == "INVITE";
    // NOTE: these addresses are *not* to be taken as real IP-addresses but rather as place holders for the real values to be put in during test execution. This is needed so that the same test cases could be executed in different test execution environments.
    require invite.headers.from == "Alice <sip:alice@wonderland.com>";
    require invite.headers.to == "Bob <sip:bob@wonderland.com>";
    require invite.headers.via == "SIP/2.0/UDP server.wonderland.com;" + "branch=z9hG4bKjkshdyff";
}

private void requireHeaders(SIPResponse response, SIPRequest responseTo) {
    require response.headers == responseTo.headers;
}

private SIPRequest originalRequest;
private float timerA;
private float timerB;
private float timerD;
private final float T1 = 0.5;
SystemBlock.cqa

/*
* The INVITE Transaction Client has two interfaces one towards the
* "Transaction User" and one towards the network. Both of these
* can receive and send messages.
*/

system {
  // Transaction User
  Inbound  tuIn: SIPRequest;
  Outbound tuOut: SIPResponse, TransportError, Timeout;

  // Transport side
  Inbound transportIn: SIPResponse, TransportError;
  Outbound transportOut: SIPRequest;
}

// initial INVITE message example

INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKkjhdyff
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=88sja8x
Max-Forwards: 70
Call-ID: 987asjd97y7atg
CSeq: 986759 INVITE

// with corresponding successful ACK

ACK sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bKkjhdyff
To: Bob <sip:bob@biloxi.com>;tag=99sa0xk
From: Alice <sip:alice@atlanta.com>;tag=88sja8x
Max-Forwards: 70
Call-ID: 987asjd97y7atg
CSeq: 986759 ACK

record /*MESSAGE*/ SIPRequest
{
  String method;
  String requestURI;
  CommonHeaders headers;
}

// SIP header fields that are common to requests and responses.
// Notice an important point about modeling for test and the level of
// abstraction to use: we only use abstract values in header fields in our
// model. In other words, we only model *to the extent* that is necessary
// to express the things that are in the scope of the functionality and
// requirements specified! The part of the RFC we are interested in
// only talks about these fields in the context of ACK construction
// by saying that the values in the ACK (whatever they maybe) must match
// those in the original INVITE and the response being acknowledged
// --> although we *could* put in all the nitty-gritty details of these
// header fields (== to break e.g. from and to all the "atomic" components
// that they are made of), we will not do it because for our purposes it's
// not
// necessary!

record CommonHeaders
{
  String callID;
HeaderCSeq cseq;
String from;
String to;
String via;
}

record HeaderCSeq {
    String seqNo;
    String method;
}

record /*MESSAGE*/ SIPResponse {
    int statusCode;
    CommonHeaders headers;
}

record Timeout {
    // Which timeout was it?
    String timerName;
}

record TransportError {
}

Main.cqa

void main() {
    INVITEClientTransaction ct = new INVITEClientTransaction();
    ct.start();
}
### Processing Model

1. `after(timerE)` / `sendNonInvite(originalRequest);`
2. `requirement "17.1.2.2q retransmission message";`
3. `timerE = T2;`
4. `requirement "17.1.2.2r the timer is reset but with a value of T2";`

### Trying Model

1. `after(timerE)` / `sendNonInvite(originalRequest);`
2. `requirement "17.1.2.2q retransmission message";`
3. `timerE = T2;`
4. `requirement "17.1.2.2r the timer is reset but with a value of T2";`
NonInviteClientTransaction.java

```java
class NonInviteClientTransaction extends StateMachine {
    public NonInviteClientTransaction()
    {
        timerE = T1;
        timerF = 64*T1;
        timerK = T4;
    }

    protected void sendNonInvite(SIPRequest NonInvite)
    {
        require NonInvite.method == "Non-INVITE";
        require NonInvite.requestURI == "sip:bob@wonderland.com";
        requireHeaders(NonInvite);
        originalRequest = NonInvite;
        transportOut.send(NonInvite);
    }

    protected void sendTimeout(String timerName)
    {
        Timeout t;
        t.timerName = timerName;
        tuOut.send(t);
    }

    protected void forwardResponseToTu(SIPResponse resp)
    {
        requireHeaders(resp, originalRequest);
        tuOut.send(resp);
    }

    protected void forwardErrorToTu(TransportError er)
    {
        tuOut.send(er);
    }

    protected boolean statusInRange(int statusCode, int lowerBound, int upperBound)
    {
        return statusCode >= lowerBound && statusCode <= upperBound;
    }

    private void requireHeaders(SIPRequest NonInvite)
    {
        require NonInvite.headers.callID == "1";
        require NonInvite.headers.cseq.seqNo == "1";
        require NonInvite.headers.cseq.method == "Non-INVITE";
        require NonInvite.headers.from == "Alice <sip:alice@wonderland.com>";
        require NonInvite.headers.to == "Bob <sip:bob@wonderland.com>";
        require NonInvite.headers.via == "SIP/2.0/UDP server.wonderland.com;" + "branch=z9hG4bKk1shdyff";
    }
}
```
private void requireHeaders(SIPResponse response, SIPRequest responseTo) {
    require response.headers == responseTo.headers;
}

protected float min(float a, float b) {
    if (a < b)
        return a;
    return b;
}

private SIPRequest originalRequest;
private float timerE;
private float timerF;
private float timerK;
private final float T1 = 0.5;
private final float T2 = 4;
private final float T4 = 5;

SystemBlock.cqa

system {
    Inbound tuIn: SIPRequest;
    Outbound tuOut: SIPResponse, TransportError, Timeout;
    Inbound transportIn: SIPResponse, TransportError;
    Outbound transportOut: SIPRequest;
}

record SIPRequest {
    String method;
    String requestURI;
    CommonHeaders headers;
}

record CommonHeaders {
    String callID;
    HeaderCSeq cseq;
    String from;
    String to;
    String via;
}

record HeaderCSeq {
    String seqNo;
    String method;
}

record SIPResponse {
Appendix B

```java
int statusCode;
CommonHeaders headers;
}

record Timeout
{
    String timerName;
}

record TransportError
{
}

Main.cqa

void main() {
    NonInviteClientTransaction ct = new NonInviteClientTransaction();
    ct.start();
}
```
InviteServerTransaction.java

class InviteServerTransaction extends StateMachine {
    public InviteServerTransaction() {
        timerG = T1;
        timerH = 64*T1;
        timerI = T4;
    }

    protected void sendInvite(SIPRequest invite) {
        require invite.method == "INVITE";
        require invite.requestURI == "sip:bob@wonderland.com";
        requireHeaders(invite);
        originalRequest = invite;
        tuOut.send(invite);
    }

    protected void sendTimeout(String timerName) {
        Timeout t;
        t.timerName = timerName;
        tuOut.send(t);
    }

    protected void forwardResponseToTransport(SIPResponse resp) {
        requireHeaders(resp, originalRequest);
        recentResp = resp;
        transportOut.send(resp);
    }
}
protected void forwardErrorToTu(TransportError er) {
    tuOut.send(er);
}

protected boolean statusInRange(int statusCode, int lowerBound, int upperBound) {
    return statusCode >= lowerBound && statusCode <= upperBound;
}

private void requireHeaders(SIPRequest invite) {
    require invite.headers.callID == "1";
    require invite.headers.cseq.seqNo == "1";
    require invite.headers.cseq.method == "INVITE";
    require invite.headers.from == "Alice <sip:alice@wonderland.com>";
    require invite.headers.to == "Bob <sip:bob@wonderland.com>";
    require invite.headers.via == "SIP/2.0/UDP server.wonderland.com;" +
        "branch=z9hG4bKjkshdyff";
}

private void requireHeaders(SIPResponse response, SIPRequest responseTo) {
    require response.headers == responseTo.headers;
}

// This is the new method in this class for generate the 100 response.
protected void forwardTryingResponseToTransport() {
    SIPResponse tryingResp;
    tryingResp.statusCode = 100;
    tryingResp.headers = originalRequest.headers;
    recentResp = tryingResp;
    transportOut.send(tryingResp);
}

protected float min(float a, float b) {
    if (a < b) return a;
    return b;
}

private SIPRequest originalRequest;
// This is the new parameter
private SIPResponse recentResp;
private float timerG;
private float timerH;
private float timerI;
const float T1 = 0.5;
const float T2 = 4;
const float T4 = 5;
}

SystemBlock.cqa

system {
    Inbound tuIn: SIPResponse;
    Outbound tuOut: SIPRequest, TransportError, Timeout;

    Inbound transportIn: SIPRequest, TransportError;
    Outbound transportOut: SIPResponse;
}

record SIPRequest {
    String method;
    String requestURI;
    CommonHeaders headers;
}

record CommonHeaders {
    String callID;
    HeaderCSeq cseq;
    String from;
    String to;
    String via;
}

record HeaderCSeq {
    String seqNo;
    String method;
}

record SIPResponse {
    int statusCode;
    CommonHeaders headers;
}

record Timeout {
    String timerName;
}

record TransportError {
}

Main.cqa

void main() {
    InviteServerTransaction st = new InviteServerTransaction();
    st.start();
}
NonInviteServerTransaction.java

```java
public class NonInviteServerTransaction extends StateMachine {
    // Class implementation details...
}
```
public NonInviteServerTransaction()
{
    timerJ = 64*T1; // == 32
}

protected void sendNonInviteACK(SIPRequest nonInvite)
{
    require nonInvite.method == "NON-INVITE and NON-ACK";
    require nonInvite.requestURI == "sip:bob@wonderland.com";
    requireHeaders(nonInvite);
    originalRequest = nonInvite;
    tuOut.send(nonInvite);
}

protected void forwardResponseToTransport(SIPResponse resp)
{
    requireHeaders(resp, originalRequest);
    recentResp = resp;
    transportOut.send(resp);
}

protected void forwardErrorToTu(TransportError er)
{
    tuOut.send(er);
}

protected boolean statusInRange(int statusCode, int lowerBound, int upperBound)
{
    return statusCode >= lowerBound && statusCode <= upperBound;
}

private void requireHeaders(SIPRequest invite)
{
    require invite.headers.callID == "1";
    require invite.headers.cseq.seqNo == "1";
    require invite.headers.cseq.method == "INVITE";
    // NOTE: these addresses are *not* to be taken as real
    // IP-addresses but rather as place holders for the real values
    // to be put in during test execution. This is needed so that
    // the same test cases could be executed in different test
    // execution environments.
    require invite.headers.from == "Alice <sip:alice@wonderland.com>";
    require invite.headers.to == "Bob <sip:bob@wonderland.com>";
    require invite.headers.via == "SIP/2.0/UDP server.wonderland.com;" +
        "branch=z9hG4bKkJshdyff";
}

private void requireHeaders(SIPResponse response, SIPRequest responseTo)
{
require response.headers == responseTo.headers;
}

private SIPRequest originalRequest;
// This is the new parameter
private SIPResponse recentResp;
private float timerJ;
const float T1 = 0.5;

}

SystemBlock.cqa

system {
    Inbound tuIn: SIPResponse;
    Outbound tuOut: SIPRequest, TransportError;

    Inbound transportIn: SIPRequest, TransportError;
    Outbound transportOut: SIPResponse;
}

record SIPRequest
{
    String method;
    String requestURI;
    CommonHeaders headers;
}

record CommonHeaders
{
    String callID;
    HeaderCSeq cseq;
    String from;
    String to;
    String via;
}

record HeaderCSeq
{
    String seqNo;
    String method;
}

record SIPResponse
{
    int statusCode;
    CommonHeaders headers;
}

record Timeout

void main() {
    NonInviteServerTransaction nst = new NonInviteServerTransaction();
    nst.start();
}
Appendix C: FTP Case Study Model

C1: ABOR, HELP, MODE command
class FTPModel_One extends StateMachine {
    private FTPcmd originalcmd;
    public FTPModel_One()
    {
    }

    protected void saveCmd(FTPcmd c)
    {
        originalcmd=c;
    }

    protected boolean codeInRange(int Code, int lowerBound, int upperBound)
    {
        return Code >= lowerBound && Code <= upperBound;
    }

    protected void giveResponse(String s)
    {
        Response r;
        r.status=s;
        userOut.send(r);
    }
}

System.cqa

class FTPModel_One extends StateMachine {
    private FTPcmd originalcmd;
    public FTPModel_One()
    {
    }

    protected void saveCmd(FTPcmd c)
    {
        originalcmd=c;
    }

    protected boolean codeInRange(int Code, int lowerBound, int upperBound)
    {
        return Code >= lowerBound && Code <= upperBound;
    }

    protected void giveResponse(String s)
    {
        Response r;
        r.status=s;
        userOut.send(r);
    }
}

Main.cqa

void main() {
    FTPModel_One ct = new FTPModel_One();
    ct.start();
}
C2: LIST, RETR, STOR command
class FTPModel_Two extends StateMachine
{
    private FTPcmd current;
    public FTPModel_Two()
    {
    }

    protected void saveCmd(FTPcmd c)
    {
        current = c;
    }

    protected boolean codeInRange(int Code, int lowerBound, int upperBound)
    {
        return Code >= lowerBound && Code <= upperBound;
    }

    protected void giveResponse(String s)
    {
        Response r;
        r.status = s;
        userOut.send(r);
    }
}

System.cqa

system {
    Inbound userIn: FTPcmd;
    Inbound tuIn: replyCode;

    Outbound userOut: Response;
}

record FTPcmd
{
    String cmd;
}
record Response
{
    String status;
}
record replyCode
{
    int code;
}

Main.cqa

void main()
{
    FTPModel_Two ct = new FTPModel_Two();
    ct.start();
}
C3: RNFR and RNTO sequence command
class FTPModel_Three extends StateMachine {
    private FTPcmd current;
    public FTPModel_Three()
    {
    }

    protected boolean codeInRange(int Code, int lowerBound, int upperBound)
    {
        return Code >= lowerBound && Code <= upperBound;
    }
    protected void giveResponse(String s)
    {
        Response r;
        r.status = s;
        userOut.send(r);
    }
}

System.java

system {
    Inbound userIn: FTPcmd;
    Inbound tuIn: replyCode;

    Outbound userOut: Response;
}
record FTPcmd
{
    String cmd;
}
record Response
{
    String status;
}
record replyCode
{
    int code;
}

Main.java

void main() {
    FTPModel_Three ct = new FTPModel_Three();
    ct.start();
}
C4: REST and APPE sequence command
class FTPModel_Four extends StateMachine {
    //private FTPcmd current;
    public FTPModel_Four() {
    }

    protected boolean codeInRange(int Code, int lowerBound, int upperBound) {
        return Code >= lowerBound && Code <= upperBound;
    }

    protected void giveResponse(String s) {
        Response r;
        r.status=s;
        userOut.send(r);
    }
}

System.cqa

system {
    Inbound userIn: FTPcmd;
        Inbound tuIn: replyCode;

    Outbound userOut: Response;
}

record FTPcmd {
    String cmd;
}

record Response {
    String status;
}

record replyCode {
    int code;
}

Main.cqa

void main() {
    FTPModel_Four ct = new FTPModel_Four();
    ct.start();
}
C5: User, Pass and Acc sequence command

FTPModel_Five.java
class FTPModel_Five extends StateMachine {
    //private FTPcmd current;
    public FTPModel_Five() {
        
    }

    protected boolean codeInRange(int Code, int lowerBound, int upperBound) {
        return Code >= lowerBound && Code <= upperBound;
    }

    protected void giveResponse(String s) {
        Response r;
        r.status=s;
        userOut.send(r);
    }
}

System.cqa

system {
    Inbound userIn: FTPcmd;
    Inbound tuIn: replyCode;
    Outbound userOut: Response;
}

record FTPcmd {
    String cmd;
}

record Response {
    String status;
}

record replyCode {
    int code;
}

Main.java

void main() {
    FTPModel_Five ct = new FTPModel_Five();
    ct.start();
}