MASTER OF SCIENCE THESIS

APARNA VIJAYA

Supporting the automatic test case and oracle generation using system models

Thesis supervisor: Antonio Cicchetti
Thesis examiner: Mikael Sjödin

June, 2009
Acknowledgment

Making this master’s thesis has in many ways been a great journey for me; especially having hands on experience in various modeling and model based testing tools.

First of all I would like to thank my supervisor Antonio Cicchetti for giving me this opportunity as well as excellent guidance and much needed pointers and advices during the course of the thesis.

I would like to thank Daniel Flemström, Sigrid Eldh and Malin Rosqvist whose discussions and input have been inspiring.

Last but not the least I would like to express my sincere gratitude to MDH, Ericsson and TATA Consultancy Services (TCS) for giving me this opportunity for pursuing the Master degree.

Thank you all.
Abstract

In the current scenario, the software systems which are developed are becoming highly complex. So there arises a need for providing a better way for correctness and efficiency for them. In November 2000, the Object Management Group (OMG) made public the MDA (Model-Driven Architecture) initiative, a particular variant of a new global trend called MDE (Model Driven Engineering). MDE is presently making several promises about the potential benefits that could be reaped from a move from code-centric to model-based practices which may help in this era of rapid platform evolution.

In model-based testing the tester designs a behavioral model of the system under test. This model is then given to a test generation tool that will analyze the model and produce the test cases based on different coverage criteria. These test cases can be further run on the system or test harness in an automatic or manual way.

This master thesis investigates the various approaches that can be used for automatic test case generation from the behavioral model. The advantages with these new approaches are that it gives a better overview of test cases, better coverage of the model and it helps in finding errors or contradictions in minimum time.
# Table of Contents

1. Introduction ..................................................................................................................... 1  
   1.1 Motivation for selecting the topic ............................................................................. 1  
   1.2 Goals .......................................................................................................................... 1  
      1.2.1 Challenges ........................................................................................................... 1  
   1.3 Disposition ................................................................................................................. 2  
2. Technical Information...................................................................................................... 3  
   2.1 Introduction to Model Driven Development ............................................................. 3  
   2.2 Introduction to Model based testing ......................................................................... 5  
   2.3 Importance of Model based testing .......................................................................... 7  
   2.4 Different scenarios in MBT ........................................................................................ 8  
3. Comparison and Analysis of Existing MBT Tools.............................................................. 9  
4. Model Based Testing At Ericsson...................................................................................12  
5. Approach for automatic test case generation ...............................................................14  
   5.1 Different Coverage Criteria......................................................................................14  
      5.1.1 Structural model coverage................................................................................15  
      5.1.1.3 Transition-Based ............................................................................................ 16  
      5.1.1.4 Decision coverage .......................................................................................... 16  
      5.1.2 Data coverage criteria.......................................................................................16  
      5.1.3 Requirements-based criteria ............................................................................17  
6. Existing Methods............................................................................................................17  
   6.1 The Combinatorial Design Paradigm ........................................................................17  
   6.2 Usecase approach .................................................................................................... 19  
      6.2.1 Adding Contracts to the Use Cases ................................................................... 20  
   6.3 Approach in LEIRIOS Tool......................................................................................... 21  
7. Overview of Bridgepoint, TTCN and Tefkat ................................................................... 23  
8. Preferred Approaches for automatic test generation ...................................................25  
   8.1 Method for supporting test case generation using “Tefkat” from class diagrams in  
      Bridgepoint ....................................................................................................................25  
      8.1.1 Tefkat transformation.......................................................................................26  
      8.1.2 MOF Script ........................................................................................................ 26
### Definitions and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence Framework</td>
</tr>
<tr>
<td>ATS</td>
<td>Abstract Test Suites</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DSML</td>
<td>Domain Specific Modeling Language</td>
</tr>
<tr>
<td>ETS</td>
<td>Executable Test Suites</td>
</tr>
<tr>
<td>MBT</td>
<td>Model Based Testing</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model Driven Development</td>
</tr>
<tr>
<td>MOF</td>
<td>MetaObject Facility</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>TPTP</td>
<td>Eclipse Test &amp; Performance Tools Platform</td>
</tr>
<tr>
<td>TTCN</td>
<td>Testing and Test Control Notation</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UCTS</td>
<td>Use Case Transition System</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>CTL</td>
<td>Computation Tree Logic</td>
</tr>
<tr>
<td>UTG</td>
<td>UML Behavioral Test Case Generator</td>
</tr>
</tbody>
</table>
List of figures

Figure 1  Relationship between a system and a model
Figure 2  Basic MBT process
Figure 3  Model based testing technique-steps
Figure 4  Data flow hierarchy
Figure 5  Transition hierarchy
Figure 6  Method for testing based on requirements
Figure 7  Bridgepoint tool
Figure 8  MOF Transformation
Figure 9  Steps for automatic test generation using class diagrams
Figure 10 Steps for automatic test generation using sequence diagrams
Figure 11  Work process
List of tables

Table 1  Comparison of tools

Table 2  Comparison of tools using ISO/IEC 9126 standard
1. Introduction

1.1 Motivation for selecting the topic

Model Driven Development (MDD) is increasingly gaining the attention of both industry and research communities. It has generated considerable interest among IT organizations looking for a more rigorous and productive way of developing systems.

First and foremost my interest and passion in learning about the model based development led me to choose this topic. From the courses I took during studies in the past, I was always interested in UML and have a strong base in UML aspects. MDD has captured my attention and curiosity from the day I started reading articles on that and I have been really interested in exploring more about it. I think research in this area has developed my own potential, engage me to work with this thesis with full passion and enthusiasm. I also think this work will help me in my future career.

Moreover the collaboration with the industrial partner Ericsson, a leading telecom company, where Model Driven Development is an approach which is widely used also made me interested in this area.

1.2 Goals

The goal is to design a test generator that addresses the whole test design problem from choosing input values and generating sequence of operation calls to generate executable tests that include verdict information. A prototype implementation of the test generator would also be provided.

1.2.1 Challenges

This task is challenging as it includes more than just generating the test input data or test sequences that calls system under test (SUT). To generate tests with oracle, the test generator must know enough about the expected behavior of SUT to be able to check SUT output values.
1.3 Disposition

This report has been structured to suit both readers that are well informed on the concepts of MBT and readers who have never heard of these before. The general concepts of the Model Driven Development, Model Based Testing and the approaches that are existing in model based testing are described in separate chapters so that the reader can skip or read these according to his or her own level of expertise. The first three chapters are only meant to give a brief introduction to the general concepts of the subjects, for further details the author recommends tracing back the sources for this report.

Chapter one describes the motivation for selecting this thesis topic and the goals of the thesis.

Chapter two concentrates on the basic concepts of model driven development and model based testing.

Section three contains a comparative study and analysis done on various model based testing tools.

Fourth chapter gives a short overview of the model based testing approach done at Ericsson.

Chapters five and six describe various approaches used in model based testing. Some for the existing techniques used for automatic test generation are also discussed in this section.

Chapter seven gives an overview about the tools and approaches used in the upcoming sections.

Chapter eight includes the two approaches proposed for automatic test case and oracle generation using models built in bridgepoint.

Chapter nine holds the details about the work process used in this entire thesis.

The results of the thesis and the future enhancements are discussed in the further chapters.
2. Technical Information

2.1 Introduction to Model Driven Development

In November 2000, the OMG made public the MDA (Model-Driven Architecture) initiative, a particular variant of a new global trend called MDE (Model Driven Engineering) [15].

MDA is one of the main fields of Model-Driven Engineering (MDE). MDA is a specific MDE deployment effort around industrial standards such as MOF, UML and so on. OMG’s Model-Driven Architecture is a strategy towards interoperability across heterogeneous middleware platforms through the reuse of platform independent designs. MDD is a development practice where high-level, agile, and iterative software models are created and evolved as software design and implementation takes place. In MDD there is a separation of the model from the code, the user works on a platform independent model, selects the specific target platform, and the tool generates the code [15].

"Everything is a model" is the current driving principle for the MDE/MDA [15].

The major reason for software crisis now is the rapid platform evolution and complexity management. Systems are becoming more complex by:

- Increasing Volume
  - Data
  - Code
  - Aspects (functional and non-functional)
- Increasing evolutivity:
  - of the business part (globalization, reorganizations, increasing competition)
  - of the execution platform
- Increasing heterogeneity
  - languages and paradigms
  - data handling and access protocols
  - operating systems and middleware platforms
  - technologies

The OMG’s Model Driven Architecture (MDA) initiative is aimed at using modeling and meta-modeling to drive the design and implementation of
distributed systems. MDA aims to solve the complexity management problem with a set of coordinated DSLs (Domain Specific Languages).

According to OMG, MDA:
- Decouple neutral business models from variable platforms
- From code-centric to model-centric approaches
- "Transformations as assets"

The model represents a complex structure that represents a design artifact, such as a relational schema, object-oriented interface, UML model, XML DTD, web-site schema, semantic network, complex documents, or software configuration.

![Figure 1: Relationship between a system and a model](image)

A model is a representation of a system. A model is written in the language of its unique meta-model. A meta-model is written in the language of its unique meta-meta-model.

The purpose of a model is always to be able to answer some specific sets of questions in place of the system, exactly in the same way the system itself would have answered similar questions. Each model represents a given aspect of the system.

A meta-model defines a consensual agreement on how elements of a system should be selected to produce a given model. That is, the correspondence between a system and a model is precisely defined by a meta-model. Each meta-model defines a DSL. Each metamodel is used to specify which particular "aspect" of a system should be considered to constitute the model.

Modeling can help [34]:

---

4
• Help in specifying how the system we want shall be
• Allow the specification of the structure and behavior of the system
• Provide guidelines on how to build the system
• Analyze system characteristics, even before the system is built

### 2.2 Introduction to Model based testing

The increase in testing effort is growing exponentially with system size and testing cannot keep pace with the development of complexity and size of systems. Model based testing has potential to combine the testing practice with the theory of formal methods. It promises better, faster, cheaper testing by:

- Algorithmic generation of tests and test oracles
- Formal and unambiguous basis for testing
- Measuring the completeness of tests
- Maintenance of tests through model modification

If source code is not available or the implementation has not started, test cases can be produced out of a model [34]. This is model based testing. The first and foremost is that we need a behavioral model of the system like state diagrams or sequence diagrams. It can be either through formal notation (formal testing) or diagrammatic approach (UML based testing). Model-based testing is software testing in which test case is derived in whole or in part from a model that describes some (usually functional) aspects of the system under test.

Model-based testing is a new and evolving technique for generating a suite of test cases from requirements. Testers using this approach concentrate on a data model and generation infrastructure instead of hand-crafting individual tests [28].

The MBT process begins with requirements. A model for user behavior is built from requirements for the system. Those building the model need to develop an understanding of the system under test and of the characteristics of the users, the inputs and output of each user, the conditions under which an input can be applied, etc. The model is used to generate test cases, typically in an automated fashion. The specification of test cases should include expected outputs. The model can generate some information on outputs, such as the expected state of the system. Other information on expected outputs may come from somewhere else, such as a test oracle. The system is run against the generated tests and the outputs are compared with the expected outputs. The failures are used to identify bugs in the system. The test data is also used to make decisions, for example, on whether testing should be terminated and the system released. [33].

The basic process of MBT can be described by the figure below. An abstract model is created from the system under test (SUT); the model provides abstract test suites (ATS) which in turn is converted to executable test suites (ETS) which are performed on the SUT [42].
Model-based testing depends on three key technologies: the notation used for the data model, the test-generation algorithm, and the tools that generate supporting infrastructure for the tests (including expected outputs). Unlike the generation of test infrastructure, model notations and test generation algorithms are portable across projects [28].

Figure 2: Basic MBT process [42]

Figure 3: Model based testing technique-steps [33]
They key advantage of this technique is that the test generation can systematically derive all combination of tests associated with the requirements represented in the model to automate both the test design and test execution process.

Model-based approaches, leverages models to support requirement defect analysis and to automate test design. Model checking can ensure properties, like consistency, are not violated. In addition a model helps refine unclear and poorly defined requirements. Once the models are refined, tests are generated to verify the SUT (System under Test). Eliminating model defects before coding begins, automating the design of tests, and generating the test drivers or scripts results in a more efficient process, significant cost savings, and higher quality code. Some other advantages of this approach include [29]:

- All models can use the same test driver schema to produce test scripts for the requirements captured in each model.
- When system functionality changes or evolves, the logic in the models change, and all related test are regenerated using the existing test driver schema
- If the test environment changes, only the test driver schema needs modification. The test drivers associated for each model can be re-generated without any changes to the model.

2.3 Importance of Model based testing

Model based testing generally does not make the whole test process automated but help the testers get over the tricky parts like test case generation and trouble shooting. Model based testing can result in the following benefits:

1. Lower time consumption, lower cost and better quality due to automation of big parts of the testing tasks [1, 3]
2. Better communication between teams and individuals in using a model of user behavior that can serve as point of reference to all (“everyone speaks the same language”) [3]
4. Early exposure of ambiguities in specification and design [1]
5. Capability of automatically generating many non-repetitive and useful tests [1]
6. Eases the updating of test suites for changed requirements [1]
7. Help defining poorly written requirements to ensure good testability of the SUT [5]

8. Comprehensive tests as the test case generation tool traverse the model in a thorough way so that many test cases that may not come up during manual testing. Model-based testing tools often have features to test boundary values where statistically many faults often occur.

9. Traceability is achieved which implies that it is possible to relate a requirement to the model and a certain test case to the model and the requirement specification. The traceability helps when you should justify why a specific test case was generated from the model and what requirements have been tested.

2.4 Different scenarios in MBT

In this section we discuss four scenarios that show the different ways in which models can be used for test case generation and code generation. The first scenario concerns the process of having one model for both code and test case generation. The second and third scenarios concern the process of building a model after the system it is supposed to represent; here we distinguish between manual and automatic modeling. The last scenario discusses the situation where two distinct models are built [37].

In the first scenario, a common model is used for both code generation and test case generation. Testing always involves some kind of redundancy: the intended and the actual behavior. When a single model is used for code generation and test case generation, the code (or model) would be tested against itself [37].

Our second scenario is concerned with extracting models from an existing system. The process of building the system is conventional: somehow, a specification is built, and then the system is hand coded. Once the system is built, one creates a model manually or automatically, and this model is then used for test case generation. We run into the same problem of not having any redundancy [37].

The third case is manually building the model for test case generation even when the system is built on top of a different specification. Depending on how close the interaction between the responsible for specification and model is, there will in general be the redundancy [37].

The last case specifies the use of two independent models, one for test case generation, and one for development. This model may or may not be used for the automatic generation of code [37].
3. Comparison and Analysis of Existing MBT Tools

The major idea behind MBT is that any test case that is defined is just a traversal of the model of the system—either an explicitly defined one or the one which is planned to be developed. MBT is therefore especially useful for automatic generation of test sequences from the defined model. Various graph theory algorithms can be employed to walk this graph (e.g. shortest path, N-states, all-states, all-transitions etc.). The best part is that when you are using a good MBT tool, there is no need to dig into all this graph theory all the time. Simply define the states, actions, transitions etc. for your model and click a button to start generating tons of interesting test cases [35].

There was a hands-on experience with several commercial and academic modelling tools to become familiarize with the modelling approaches and few of them even supported model based testing.

The different tools were analysed by listing down all the available tools in market for model based testing (around 15 different kinds). We classified them as commercial and academic tools based on their availability. The trial versions of possible commercial tools were downloaded; in some other cases the corresponding organizations were contacted to obtain the license for the tools. Finally there were around seven different tools and started modelling a simple communication protocol. Some of the tools supported UML diagrams and it was comparatively easy to start using them. But in some cases like Spec Explorer and JUMBL we had to learn some new language like spec# in order to model the system. It was quite a challenging task because of the limited access to user manuals. Few tools supported automatic generation of test cases whereas in some tools like UPPAAL the user manually specifies the test cases. We also tried out the options of interchanging models and test cases between the various tools. Unfortunately due to the difference in the formats supported by each tool, this task became impossible.

Comparative studies of the different tools are represented in the following table.

Table 1: Comparison of tools

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Type</th>
<th>Modelling Language</th>
<th>Usability</th>
<th>Automatic generation of test cases</th>
<th>Export models to other formats</th>
<th>Model Simulation</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPAAL</td>
<td>Academic</td>
<td>UML</td>
<td>Yes</td>
<td>No</td>
<td>Can export to XML file</td>
<td>Yes</td>
<td>Windows</td>
</tr>
<tr>
<td>Conformiq</td>
<td>Commercial</td>
<td>QML</td>
<td>Yes</td>
<td>Yes</td>
<td>Export &amp; Import XMI files</td>
<td>No</td>
<td>Windows/Linux</td>
</tr>
<tr>
<td>QTRONIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spec Explorer</td>
<td>Research</td>
<td>SPEC # &amp; ASML</td>
<td>Yes</td>
<td>Yes</td>
<td>Can export to XML file</td>
<td>No</td>
<td>Windows</td>
</tr>
<tr>
<td>Model</td>
<td>Academic</td>
<td>Java</td>
<td>No</td>
<td>Yes</td>
<td>Can</td>
<td>No</td>
<td>Windows</td>
</tr>
</tbody>
</table>
One of the most interesting analyses found during this process is that even though some tools using the same modeling language (for e.g.: UPPAAAL and Cow Suite uses UML), models cannot be imported/exported between these tools.

For specifying quality criteria for testing tools, the ISO/IEC 9126 standard was taken into account. It includes 6 characteristics for software product quality, 3 vendor qualification details, out of which only 1 vendor characteristic was taken into consideration. It is represented in the following table.

Table 2: Comparison of tools using ISO/IEC 9126 standard

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Functionality</th>
<th>Reliability</th>
<th>Usability</th>
<th>Efficienc</th>
<th>Maintain-</th>
<th>Portability</th>
<th>Licensing and pricing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise</td>
<td>Suitability:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architect</td>
<td>Supports only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>modeling and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>not MBT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spec Explorer</td>
<td>Interoperability: Databases parts of states, transitions and test suites can be exported to an XML file but not the State machine diagram as whole. So interoperability is not achieved in this tool.</td>
<td>Fault</td>
<td>Easy to</td>
<td>Analyzability: Yes</td>
<td>Installability: Simple and Easy to install instructions are provided.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tolerance: Tool does not support this. Recoverability: If the application crashes it can be recovered as data is stored locally.</td>
<td>Tolerance</td>
<td>operate with good GUI. But application does not support lower versions Microsoft word 2007.</td>
<td>Easy to analyze the application behavior.</td>
<td>Installability: Open Source</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model J Unit</td>
<td>There is a provision for user to add his new test generation algorithms.</td>
<td>Recoverability: If the application crashes it can be recovered as data is stored locally.</td>
<td>GUI that is downloaded from internet is not working</td>
<td>Supports all phases of MBT</td>
<td>Since GUI is not working we could not analyze the entire application.</td>
<td>Installability:</td>
<td></td>
</tr>
<tr>
<td>UPPAAL</td>
<td>Suitability: supports MBT</td>
<td>Fault tolerance:</td>
<td>Yes, graphical</td>
<td>Analyzability: Yes</td>
<td>Installability: Easy</td>
<td>Installability: Open source and</td>
<td></td>
</tr>
<tr>
<td>Tool</td>
<td>Suitability: supports MBT</td>
<td>Interoperability: models nor test cases cannot be interchanged between tools</td>
<td>Interoperability: models nor test cases cannot be interchanged between tools</td>
<td>Only if the model is created and simulated without any errors testing is possible. Each phase is completely depended on previous phase</td>
<td>Changeability: Yes Testability: Yes</td>
<td>Changeability: Yes Testability: Yes</td>
<td>Changeability: Yes Testability: Yes</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>Cow Suite</td>
<td>Yes, graphical user interface and drag and drop approach</td>
<td>Time, effort and resource used: less</td>
<td>Analyzability: Yes Changeability: Yes Testability: Yes</td>
<td>Analyzability: No Changeability: Yes Testability: Yes</td>
<td>Installability: easy</td>
<td>Installability: time consuming</td>
<td>Installability: Open source and academic tool</td>
</tr>
<tr>
<td>JUMBL</td>
<td>No, a new language called TTL has to be mastered for modeling</td>
<td>Time, effort and resource used: High</td>
<td>Analyzability: Yes Changeability: Yes Testability: Yes</td>
<td>Analyzability: Yes Changeability: Yes Testability: Yes</td>
<td>Installability: time consuming</td>
<td>Installability: Open source and academic tool</td>
<td>Installability: Open source and academic tool</td>
</tr>
</tbody>
</table>
4. Model Based Testing At Ericsson

Spirent testing tool which supports manual testing was used in Ericsson in certain departments mainly for functionality testing. This is a complete tool for creating and running test cases. From the test specification and the function specification the testers assigned to the functionality creates test cases. This means that the tester creates test cases directly by creating flows of state machines and specifying inputs and expected outputs. The actual comparison is done automatically and error messages are created for cases when the generated outputs differ from the expected outputs. Each test case has to be checked manually one by one to where the error has occurred. But one of the biggest problems with manual testing using the Spirent tool was during regression testing; which turned out to be very time consuming. The reason for that is Spirent checks each test case manually one at a time. During regression testing when there is something to change every time, it becomes very time consuming and tedious [41].

Conformiq together with Ericsson is now using an approach where a model of the functionality is created and it automatically generates TTCN3-code in Qtronic. A test harness was created to receive the TTCN3-code and run the tests against a SUT simulator called Vega and return the results; this was done in the TITAN-framework since this is a framework that has been used a lot on other departments at Ericsson. TITAN framework is the first fully functional TTCN-3 test tool. Regarding the test harness there was again problems with the operability since the test cases had to be checked one by one. If the problem was solved there would still be an issue in operability in the way that there is currently no way of seeing all test results at once. According to an expert consultant at Conformiq this could easily be solved by connecting the test harness back to the Qtronic tool where the results could be nicely presented to the user [41].

Now Ericsson also uses Bridgepoint from Mentor Graphics which is used to model the system. Bridgepoint supports xtUML modeling with components diagrams, class diagrams, state machines and action language. Bridgepoint also has support for model verification. It is possible to model and interact with existing code. But it doesn’t support automatic generation of tests; rather tests are manually written for a model [42].

Automated tests ensures a low defect rate and continuous progress because of the “reuse” tests in form of regression tests on evolving systems, whereas manual tests would very rapidly lead to exhausted testers.
How does working with a model affect understanding of the SUT compared to the old way of working?

The opinions on this were that the understanding of the SUT should not be affected in any great way because the creators of the model still have to read documentation and learn how the system works. In manual testing the tester gets another perspective in the way that testing is always looked at from a test case point of view. In MBT this is lost which could imply testers getting a weaker picture on what they are testing since they do not see the actual test cases [42].
5. Approach for automatic test case generation

After modeling, the system has to generate the expected inputs and outputs. The expected inputs and outputs together form so called test cases which, in manual testing, used to be restricted to the testers own imagination. Moreover manual testing techniques consume much of the time of the project life cycle.

In MBT the test cases are generated automatically by using a random algorithm or a more structured approach. The approach for test case generation depends on the complexity of the model at hand.

The expected outputs are often created by the same functionality that compares expected output to the actual output, namely an “oracle”. Since more than thousands of test cases are generated automatically using MBT approach, it becomes difficult and time consuming to compare their outputs with the expected output. This leads to the creation of an “oracle” that automatically draws expected outputs. This is because of the volume of the test cases but also because of the fact that the test suites (many test cases in one area) do not remain static. This is solved depending on the system at hand but often a test case is considered to pass if it falls into a predefined range. The model helps to create suites of test cases which cannot be used directly on the SUT since they are on the wrong abstraction level. Instead executable test cases have to be created and for this purpose some code is generated in what is usually called a test harness.

5.1 Different Coverage Criteria

It is the model-based testing tool that generates the test cases but the criteria for test case selection is determined by the tool manufacturer and the end user. MBT is a black-box testing technique and the coverage criterion specifies how well the generated test cases traverse the model. Test cases are usually generated even before the actual implementation of the SUT has started. Since the source code coverage of the SUT cannot be measured, the measurement of statement and branch coverage can be done when the SUT is executed with the generated test cases.

The choice of coverage criteria determines which type of algorithms that will be used by the model-based testing tool for generating the tests, how large the test suite will be, how long time it will take to generate them and which parts of the model that will be tested. When applying a coverage criterion you are saying to the tool “please try and generate a test suite that fulfills this criterion”.

14
Maybe you are requesting something that is very hard or even impossible to solve. The tool will not perform any black magic for you; it is working in a restricted domain and will do its best in accomplishing your request. A point of failing can be that some part of the model is statistical unreachable and therefore it is not possible to accomplish the criteria. There is also a possibility that the tool is not powerful enough to find a path in the model so that the criteria can be achieved to 100 percent coverage. In the case of failing criteria the tool should be able to generate some type of report that indicates which part that could not be covered so that it can be investigated more thorough [43].

Coverage-based approaches consider the model as a directed graph structure with a set of vertices and a set of edges. A large number of syntactical coverage criteria are known, e.g. a test case shall cover all vertices in the graph or all paths.

Different tools have different test selection criteria and often only a subset of all criteria’s is discussed below.

### 5.1.1 Structural model coverage criteria

Structural model coverage is mainly used to determine the sufficiency of a given test suite. A variety of metrics exists; including statement coverage, decision coverage etc. Structural model coverage has some similarities with code based coverage criteria. The similarities are the control-flow and the data-flow coverage criteria, but transition-based and UML-based coverage criteria are only a part of structural model coverage criteria [43].

#### 5.1.1.1 Control-flow

Control-flow coverage covers criteria as statements, branches, loops and paths in the model code [43].

#### 5.1.1.2 Data flow

Data flow oriented coverage criteria cover read and writes to variables [43].

![Data flow hierarchy](image)

**Figure 4: Data flow hierarchy [43]**
5.1.1.3 Transition-Based

Transition based models are built using states and transitions; in some notations like state charts it is possible to have hierarchies with states. Two methods, one is a state chart and one is a normal finite state machine (FSM) is used for this [43].

![Transition hierarchy diagram](attachment:transition_hierarchy.png)

Figure 5: Transition hierarchy [43]

5.1.1.4 Decision coverage

Each decision of the program has been tested at least once with each possible outcome. Decision coverage is also known as branch coverage or edge coverage.

5.1.1.5 Condition coverage

Each condition of the program has been tested at least once with each possible outcome.

5.1.1.6 UML-Based

The UML language is for specifying, visualizing, constructing and documenting the artifacts of the software system. UML provides a variety of diagrams that can be used to present different views of an object oriented system at different stages of the development life cycle. Decision and transition coverage can be used in UML state machines [43].

5.1.2 Data coverage criteria

The input values for a system is often very large that all possible combinations of the input cannot be always tested. Data coverage criteria are useful for selecting a good set of few data values to use as input to the system like boundary value testing and statistical data coverage [43].
5.1.3 Requirements-based criteria
When a system is modeled, it is mandatory to verify whether all the requirements have been fulfilled. The passing of all requirements ensures that the system delivers what it should [43]. A few ways in which requirement traceability can be achieved is:

- Insert the requirements directly to the behavior model so that the test generation tool can ensure that they are fulfilled.
- Use some formal expression, like logic statements that drives the test generation tool to look for something special behavioral in the model.

5.1.4 Explicit test case specifications
Explicit test case specifications can be used to guide the test generation tool to examine some certain behavior of the system. These specifications can e.g. be a use case model which specifics some interesting path of the model that should be tested carefully [43].

6. Existing Methods

This section discusses a few of the commonly used approaches or methods for model based testing.

6.1 The Combinatorial Design Paradigm
The combinatorial design tests uncover faults in the code and in the requirements that were not detected by the standard test process. The person involved in requirement gathering and product development is involved in writing the tests in this approach.

Initially the tester identifies the system under test and defines its parameters like input and configuration parameters. For example, in testing the user interface software for a screen-based application, the test parameters are the fields on the screen. Each different combination of test parameter values gives a different test scenario. Since there are often too many parameter combinations to test all possible scenarios, the tester must use some methodology for selecting a few combinations to test [44].

The tests which are generated cover all the combinations of the test parameters like pairwise, triple, or n-way combinations. Covering all pairwise combinations means that for any two parameters \( P_1 \) and \( P_2 \) and any valid values \( V_1 \) for \( P_1 \) and \( V_2 \) for \( P_2 \), there is a test in which \( P_1 \) has the value \( V_1 \) and \( P_2 \) has the value \( V_2 \) [44].

The test requirements of the system under test are specified using a set of relations. Each relation has a set of test parameters and a set of values for each parameter. The set of possible test scenarios specified by the relation is the Cartesian product of the value sets for its test parameters. The tester specifies constraints between test parameters either by using multiple relations or by prohibiting a subset of a relation's tests. For each relation, the tester specifies the degree of interaction (for example, pairwise or triple) that must be covered. The tests are generated according to this [44].

Testers usually generate test sets with either pairwise or triple coverage. A study of user interface software at Telecordia, where the combinatorial approach is used found that most field faults were caused by either incorrect single values or by an interaction of pairs of values. Pairwise coverage is sufficient for good code coverage as per the study conducted in Telecordia. The seeming effectiveness of test sets with a low order of coverage such as pairwise or triple is a major motivation for the combinatorial design approach [44].

### 6.1.1 Advantages of this approach

The combinatorial design approach differs from traditional input testing by giving testers greater ability to use their knowledge about the SUT to focus testing. Testers define the relationships between test parameters and decide which interactions must be tested. Using the combinatorial design method, testers often model the system's functional space instead of its input space [44].
Since this approach uses the Cartesian product of the various test parameters to generate the tests, the total number of tests needed to cover a specified requirement will be less. Testers can create effective and efficient test plans that can reduce the testing cost while increasing the test quality [44].

6.2 Usecase approach

This approach is used for automating the generation of system test scenarios so as to reduce the traceability problems and test case execution problems. The first step in this approach is to formulate the use cases extended with contracts based on the requirements. The objective is to cover the system in terms of statement coverage with those generated tests. UML use cases enhanced with contracts means the use cases along with their pre conditions and post conditions are specified [45].

According to Meyer’s Design by Contract idea [49], these contracts are made executable by writing them in the form of requirement-level logical expressions. Based on those more formalized, but still high level requirements, a simulation model of the use cases is defined. In this way, once the requirements are written in terms of use cases and contracts, they can be simulated in order to check their consistency and correctness. The simulation model is also used to explicitly build a model of all the valid sequences of use cases, and from it to extract relevant paths using coverage criteria. These paths are called test objectives. The test objectives generation from the use cases constitutes the first phase of our approach. The second phase aims at generating test scenarios from these test objectives [45].

Each use case is documented with a sequence diagram to automate the test scenarios generation. As a result, test scenarios that are close to the implementation is obtained [45].
Figure 6: Method for testing based on requirements [45]

The steps involved in this approach are [45]:

1) Generating test objectives from a use case view of the system using the requirement by-contract approach. These contracts are used to infer the correct partial ordering of functionalities that the system should offer (Steps (a) to (c)).

2) From the use cases and their contracts, a prototype tool (UC-System) builds a simulation model (step (b)) and generates correct sequences of use cases (step (c)) called a test objective. This simulation phase allows the requirement engineer to check and possibly correct the requirements before tests are generated from them.

3) The second phase (steps (c) to (e)) aims at generating test scenarios from the test objectives. A test scenario may be directly used as an executable test case. To go from the test objectives to the test scenarios, additional information is needed, specifying the exchanges of messages involved between the environment and the system. Such information can be attached to a given use case in the form of several artifacts: sequence diagrams, state machines, or activity diagrams.

6.2.1 Adding Contracts to the Use Cases

A use case mainly depends on the specific actors to which it is connected, and to the business level concepts it has to handle. Actors involved in
a use case can be considered as parameters of this use case and contracts are expressed in the form of pre and post conditions involving these parameters. The use case contracts are first-order logical expressions combining predicates with logical operators. The precondition expression is the guard of the use case execution. The post condition specifies the new values of the predicates after the execution of the use case. When a post condition does not explicitly modify a predicate value, it is left unchanged.

6.2.2 Advantages of this approach

The main advantage of this approach is that it reduces the ambiguities which will be caused by specifying the requirements in natural language. Here the requirements are represented as use cases.

A test scenario as a sequence diagram represents a test. Test scenarios may differ from the test cases in the fact that the test cases can be applied directly with a test driver, whereas the test scenarios may still be incomplete. The test scenarios contain the main messages exchanged between the tester and the system under test.

6.3 Approach in LEIRIOS Tool

LEIRIOS Smart Testing tool uses three behavioral UML models to design a test model. They are [53]:
- Class diagram - defines the static view of the model. It describes the abstract objects of the system and their dependencies.
- Object diagram - models the initial state and test data of the SUT.
- State-machine - to model the dynamic SUT behaviors as a finite state transition system.

Object Constraint Language is used to formally express the SUT behaviors.

From a UML behavioral test model LEIRIOS Smart Testing extracts test objectives. A test objective is a pair context and effect, where effect is the system behavior to be tested and context the condition to fire that particular behavior. This test generation engine is used to search for a path from the initial state to a test objective, and data values satisfying all the constraints along this path [53].

The LEIRIOS Smart Testing solution provides adapters and exporters to manage and/or execute the generated test cases. The test cases can be translated into test scripts in any language, to be executed on the SUT thereafter. Adapters are also provided to export test cases in various test management and execution tools. The LEIRIOS Smart Testing provides a XML exporter to automatically translate abstract test cases into a tree-structured
document. This XML file contains all the information needed to know – after each operation/event call of a test – the updated state of each object, the relationship between object, and so on [53].

This file is the support to write executable tests scripts. Each UML operation/event is encapsulated in a JScript function. These JScript methods are written in a specific JScript file: the adaptation layer. The adaptation layer is a kind of specific library which provides methods to bridge the gap between the abstract test cases and the SUT. The writing of a JScript test script for each abstract test case is one of the ways to automate test execution on the SUT [53].
7. Overview of Bridgepoint, TTCN and Tefkat

The BridgePoint UML Suite, a product of Mentor Graphics, is a set of Eclipse plug-in and is delivered as an Eclipse extension so as to integrate it with the existing Eclipse installation. Using Bridgepoint the system is modeled using UML 2.0 and thus a platform independent model is created. It also facilitates the automatic code generation from UML 2.0 models in either C or C++. Instantaneous execution of UML models is also supported [48].

![Bridgepoint tool](image)

TTCN-3 (Test and Test Control Notation version 3) is a programming language for developing tests in the telecommunications domain. TTworkbench is the graphical test development and execution environment based on the international standardized testing language TTCN-3. Once tests have been defined in TTCN-3, TTworkbench compiles them in an executable test suite.

Error! Reference source not found.

Tefkat is a java based implementation of a rule and pattern-based engine for the transformation of models. Tefkat usually has a source model and a target model. It uses an implementation of a declarative language which uses meta-models to execute transformation rules to map between the source meta-model and the target meta-model. The Tefkat transformation engine requires as input the source model, the source meta-model, the target meta-model and a set of rules. It triggers the rules one-by-one on instance(s) of the source meta-model.
(i.e. source models) and generates instance(s) of the target meta-model. The selection of test cases was performed using criteria on the XMorph language metamodel elements. For each metamodel element under test, both functional and robustness test cases were generated [50].

Eclipse Modeling Framework (EMF) is used for building models and for code generation. It contains all information about data models. It creates two models, Ecore model and Genmodel. Ecore model includes all the model information with the defined classed. Genmodel contains the information for code generation that is how the coding should be generated.
8. Preferred Approaches for automatic test generation

8.1 Method for supporting test case generation using “Tefkat” from class diagrams in Bridgepoint

The design model was constructed using the Bridgepoint builder. It includes class diagrams, sequence diagrams, state charts and so on. The shared model approach is used for test generation. The same model is used for generating source code as well as for test cases. A two layer approach for abstract tests and concrete test scripts is used for the implementation. A transformation tool or some adaptor code that wraps around the SUT can be used to translate abstract test case into an executable test script which uses certain templates or mappings for this. The advantage of this is abstract tests which can be independent of the language used to write tests and of the test environment. The coverage based test selection criteria are used in this case. That is the model is represented as a graph and the test cases covers the set of vertices (or the paths) of the graph. Test suites can be created that covers as many of the different model components (a method, a transition or a conditional branch).

Bridgepoint generates usecase diagrams, class diagrams and sequence diagrams using executable UML. The class diagram is the approach we use in Tefkat for generating tests. The class diagrams created in bridgepoint can be directly converted into an Ecore model using 3 ways [46]:

1) With EMF a new genModel from a UML model can be generated: it will create automatically the Ecore file.

2) Selecting the root element of the model and using the *menu action* Sample Ecore editor/convert to UML model (or UML editor/convert to/Ecore model if you want to convert from UML to Ecore). Then choose all concepts you want to be converted (by keeping *process* options enabled in the lists).

3) Using open source eclipse plug-in like UMLUtil and Topcased tools.

The following architecture could be used in Bridgepoint to enhance the tool to support automatic test case generation.

In this approach the source model and target model are UML models created in bridgepoint. These are converted to Ecore models using any of the approach described above. EMF created two models, the Ecore model and the genmodel. The Ecore model contains the model information with the defined
classes; the genmodel contains additional information for the code generation. The xUnit models are generated by using Tefkat transformation rules which are applied on these Ecore models. MOFScript transformation rules are used to generate Junit test cases from the xUnit model [50].

8.1.1 Tefkat transformation

Each Tefkat transformation rule produces a table and key for each class. Each class is represented in terms of a table where the datatype corresponds to a row, and each attribute corresponds to column in the corresponding table. Also there is an object ID which constitutes the key. An example of the transformation rule is [50]:

```
RULE Class2Table(c, t, k)
FORALL UMLClass c
WHERE c.kind = "persistent"
MAKE Table t, Key k
SET t.name = c.name,
    t.key = k,
    k.name = c.name;
```

8.1.2 MOF Script

The MOFScript language is an initial proposal to the OMG model-to-text. The MOFScript tool, an eclipse plug-in, is an implementation of the MOFScript model to text transformation language. MOFScript tool supports EMF metamodels and models (UML2, Ecore) as input.

Consider a transformation from source model A to text t. (A -> t)

The steps involved in this process are:

- Import or create the source metamodel for A.
- Write the MOFScript to transform A to t in the MOFScript editor
- Compile the transformation. Fix errors if any has occurred
- Load a source model corresponding to A’s metamodel.
- Execute the MOFScript in the MOFScript tool.
The transformation is executed. Output text is produced in the form of XMI format.

### MOF models

<table>
<thead>
<tr>
<th>Ecore Diagrams</th>
<th>MOF Script</th>
<th>Textual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Java, C#, XMI</td>
</tr>
</tbody>
</table>

**Figure 8: MOF Transformation**

#### 8.1.3 An example for MOFScript generation

An example to simulate a simple data link protocol is described below:

The protocol specification is as follows:

There is a Sender, a Receiver, and a Medium. After the medium receives the message from the sender, one of two actions is performed: either the message is delivered to the receiver, or it is lost. When a message is lost a time-out in the sender will occur. This time-out is represented as a communication signal sent from the medium (in case of message loss) to the sender. In case of a time-out, the sender will retransmit the message. When the receiver gets a message an acknowledgment is sent to the sender. Assume that the acknowledgment is sent directly to the sender and not through any medium. The test sender increments a shared integer variable ‘i’ whenever a message is sent. The test receiver decrements the variable ‘i’ whenever a message is received.

The class diagram for this is shown in figure 9:
The textual version of MOFScript for this example using this approach would be like:

```
RULE testMofInstanceTrue (m, s)
FORALL message m, Sender s, Medium m
MAKE sentmsg sm = 1
LINKING SenderToMedium
WITH s.sentmsg = m.recvmsg;
```

This MOFScript or rule specifies that the message has been sent successfully from the sender to the medium.

### 8.1.4 Test Generation Procedure

1. Create a UML model (class diagram) in bridgepoint.
2. Using adaptors like UMLUtil and Topcased tool, convert the UML model into an Ecore model.
3. Apply Tefkat transformation to create the metamodel from the Ecore model.

4. Apply the MOF script using the MOFScript tool engine to produce the textual format of the metamodel in XMI format.

5. A finite state machine diagram is drawn from the XMI specification.

6. Further TTCN3 test cases are generated from this model based on [51] :
   a) A test case is generated by traversing the model. Method sequences are derived from the traversed transitions. Test cases are generated in such a way that it covers the test model, in the form of state coverage, transition coverage, and path coverage.
   
   b) Data flow testing techniques are used to generate test cases. Test cases are generated according to the definitional and computational use occurrences of attributes in the model. Each attribute is classified as being defined or used. An attribute is said to be defined at a transition if the method of the transition changes the value of the attribute. An attribute is said to be used at a transition if the method of the transition refers to the value of the attribute.

7. Oracle is created based on formal specification of classes. We test a class implementation by determining whether it conforms to its formal specification.

8. Execute the test cases and analyze the results.
Figure 10: Steps for automatic test generation using class diagrams
8.1.5 Enhancements

Since a class diagram represents the static behavior of the system modeled and sequence diagram represents the dynamic behavior, a mixture of both these diagrams could create more efficient tests. So if the class diagram can also include the pre and post conditions from that, it becomes a more efficient approach.

8.2 Method for supporting test case generation from sequence diagrams in Bridgepoint

The method suggested here describes a translation of UML sequence charts to TTCN3 statements for generation of tests cases.

Sequence diagrams are used to describe dynamic system properties by means of messages and corresponding responses of interacting system components. They can also be applied for the specification of test cases.

Sequence diagram represents not only the triggering method calls, but it is possible to model desired interactions and check object states during the test run [54].

A sequence diagram consists of a relation between an output and input event from and to an environment. These can be matched to send and receive operations in TTCN-3. The asynchronous and synchronous calls are mapped to TTCN-3 non-blocking and blocking calls. TTCN-3 templates and their arguments provide easier usage of communication operations because there is no other information needed to be added to that. Thus, test cases become more understandable and maintainable.

The steps involved in this process are:

1) Build sequence diagrams in Bridgepoint.
2) Generate XMI snippets for the sequence diagrams
3) Create a control flow graph using a graph builder based on this XMI snippet.
4) Use the AI framework to decide the path for which the test case has to be generated.
5) Analyze for coverage feedback.
6) If the coverage is not sufficient, generate new tests using the AI framework again.
7) Use TTCN3 to generate test scripts.
8) Execute the tests.
9) Analyze results.

This approach proposed for generating test cases from the sequence diagrams is shown in figure below:

Figure 11: Steps for automatic test generation using sequence diagrams

The approach used here for test case generation is CTest scheme. In this we select a predicate from the tree diagram which is drawn from the sequence diagram. For each predicate which is selected, we apply a transformation to find the test data corresponding to that. This process is repeated until all the predicates are considered for test case generation [55].
All messages are labeled with conditional predicates. It can be also empty which implies that it is always true. For generating the test cases predicates are selected through a post order traversal of the tree diagram (that is leaf nodes are considered initially for test case generation) [55].

8.2.1 Example for test generation using sequence diagram approach

For the example problem statement described in section 8.1.3, the sequence diagram is specified in figure 12.

![Sequence diagram for communication protocol](image)

**Figure 12: Sequence diagram for communication protocol**

The sequence diagram given here specifies the events occurring when the sender successfully sends the message to the medium and when the message is lost (specified as time out from medium to sender). The XMI snippets are generated for this and the AI framework decides the paths which have to be covered by the tests. If a particular test covers only the successful transmission part, the coverage feedback indicates it. Then the framework specifies the tests to verify the ‘message lost’ part as well. Thus this framework tries to cover all the possible paths or transitions for creating more efficient tests.

For generating the test cases for the example mentioned in Figure 12, the predicate function would be $x_m - x_s \leq t$ where $x_s$ represents the time during which sender transmitted the message, $x_m$ is the time at which the medium receives the message and $t$ represents the time out.
Now we need to apply the function minimization on this function. For this we minimize the value of the function with respect to each input variable. Here we alter the value of \( x_s \) by decreasing /increasing its value while keeping all others constant. Every condition which is covered by a path is considered as a constraint. If a particular path is not traversed, then it implies that a constraint is not satisfied for a particular value of input variable. The minimum value is specified for the input variable while starting and later it is kept incrementing using a value which is constant and called as step size. We identify the values of \( x_s \) for which the minimization function becomes zero or negative and it represents the boundary values or test data points. This data point corresponds to values which satisfy the path conditions. By this process we can generate test cases for all predicates.

![Figure 13: Modeling of communication protocol in UPPAAL](image-url)
Figures 13 and 14 depict the view of communication protocol when modeled using Uppaal. Uppaal modeling tool is based on the theory of timed automata. The query language of Uppaal, used to specify properties to be verified is a subset of CTL (computation tree logic). Uppaal consists of a verifier where we manually specify the conditions that have to be checked (Figure 14). The main drawback of this approach is that it is time consuming as we need to specify the properties that have to be checked each and every time. There are chances that all the transitions in the model will not be checked when we are doing it manually. UPPAAL does not check for zeno behavior if it can take an infinite amount of actions in finite time [56].

In the approach specified in section 8.2, the AI framework checks whether all the possible paths are covered while test case generation. Mainly message path coverage and predicate coverage are achieved using the test cases generated by this approach. Moreover redundant test cases are less likely to occur because of the generation of test data points. Branching can be easily checked because of the generation of test cases is based on conditional predicates. That is it checks all the possible actions or transitions that occur. This also reduces the number of execution steps when the test data for the previously selected predicate was generated satisfying the current predicate [55].
9. Evaluation Method

This section aims to describe the work process as well as motivate how conclusions and results have been drawn during the evaluation project. In this chapter the workflow of the thesis is described in detail.

9.1 The Work Process

The work process during this master’s thesis builds on six phases: initiation, theory, data acquisition, analysis, implementation and closure. These represent the five different stages that a master’s thesis has to go through to complete a good work. A graphical description of the working process is given in figure 15 below.

Figure 15: Work process

9.1.1 Initiation (phase 1)
During the initiation phase the project was roughly planned, time and resource plans were created with preliminary dates on when the different milestones in the project were to be finished and how time would be spent during the master’s thesis project. The initiation phase also a time for a light study on the concepts of the master’s thesis, understood what the goals of the project are and how they could be fulfilled.

9.1.2 Defining goals (phase 2)
The general method was to read documentation about each general concept. These were based on a discussion with the supervisor of this project on what aspects to look at. Based on the study a theoretical framework was created in this project. The purpose of the theory phase was to break down the concepts into hands-on usable questions. The goal here has not been to make all the answers measurable, in fact in many cases it is more interesting to get subjective answers and opinions from the experts.
9.1.3 Data acquisition (phase 3)
During the data acquisition phase, information was gathered about how testing is performed today at Ericsson and what effects could be expected in using MBT. In this phase a sample project was tried out in some of the possible open source and academic MBT tools. Also the various approaches that are available for test case generation was studied as well their pros and cons.

9.1.4 Analysis (phase 4)
The analysis phase of this master's thesis consisted of drawing reasonable conclusions out of the results from the data acquisition phase. After a hands on experience with some of the existing MBT tools, two tables were created to compare the tools. First table is created with some of the attributes which was general and Table 2 with the quality attributes specified using ISO/IEC 9126 standard.

9.1.5 Prototype Implementation (phase 5)
A prototype implementation for the test case generation is proposed. Two different possible approaches are implemented. Analyses of these two approaches are noted.

9.1.6 Closure (phase 6)
After the analysis was done and conclusions had been drawn the project was finished and the closure phase began. In this phase the project was presented both on MDH and for Ericsson. A report summarizing the details of the thesis right from the start up is created.
10. Validity and Reliability

MBT as a method is very young on the market of software testing; therefore it is very hard to find documentation evaluating the various commercial and academic tools. This fact has of course affected the reliability and validity of this. A few suggestions needed to make an improvement on the future of MBT, especially at Ericsson are discussed with respect to the manual test approach they are following. I have mainly been working with three different examples (simple lamp example, communication protocol, real example from Ericsson) for creating models and tests in the case of each tool used. This was to ensure the efficiency and reliability of the models and tests created.
11. Analysis

Model-based testing has become an emerging trend in industry because of [54]:

(i) The need for quality assurance for increasingly complex systems
(ii) The emerging model-centric development paradigm
(iii) Increasingly powerful formal verification technology

Model-based testing relies on execution traces of behavior models, both of a system under test and its environment, at different levels of abstraction. These traces are used as test cases for an implementation: input and expected output [54].

One advantage of using models for test case description is that application specific parts are modeled with UML-diagrams and other technical parts like connection to frameworks, error handling, or communication are handled by the parameterized code generator. This allows developing models that are independent of any technology and platform. Only during the generation process platform dependent elements are added. When the technology changes, we only need to update the generator, but the application defining models as well as test models can directly be reused. Another advantage is that the production code and automatically executable tests at any level are modeled by the same UML diagrams. Therefore, developers use a single homogeneous language to describe implementation and tests [54].
12. Related Work

The work related to this paper is vast and for brevity, so I highlight here some of the most relevant one’s. The automatic generation of test cases and oracles is done in various tools in different methods. In particular, I have analyzed some of the existing tools (open source academic and research tools) and worked some examples on them.

The aim of this paper is to address the issues of creating automatic tests using models built using bridgepoint. Bridgepoint as such does not support this facility. Several different approaches have been analyzed for this. Finally two different methods have been proposed for this.

Model based generated test cases can be used to ensure that the translation (code generation) from the model to the code is proper. Manual testing techniques using these models might not be efficient as they may not cover all the possible transitions and it may be time consuming as well. Therefore many approaches have been proposed for automatic generation of test cases, some of which are mentioned in chapter 6 of this paper. The most closely related approaches to my thesis are the ones mainly using test generation using class diagrams (reference [51]) ,Model-Based Test Driven Development of the Tefkat Model-Transformation Engine (reference [50]) and Test selection criteria using FSM (reference [52]). The Ctest scheme for test case generation prototype is implemented in UML behavioral Test case generator tool (UTG) (reference [55]).
13. Results and Conclusions

Two different frameworks/methods were proposed for model based test case generation based on UML representations. These UML models are built in the bridgepoint modeling tool. The UML diagrams that are being considered here is class diagrams and sequence diagrams. Various transformation rules are applied on these models to produce the state diagrams from the XMI snippets. Further TTCN 3 is used to generate the test cases and oracles from them based on different test and coverage techniques.

These frameworks support a systematic testing process, which makes it easier to choose test cases, trace test executions, and analyze test results. Analyses reveal that this is a better way to trace the various deficiencies in the models since there are possibilities that it covers every aspect of the system (both static and dynamic, if class diagrams could include pre and post conditions). Moreover this underlying framework for translation and test generation is very flexible and easily extendable such that it can be customized for further enhancements.

13.1 Recommendations and Future Work

In the future, the framework can be modified to make it more robust and to enhance the criteria's for creating the test oracle. More efficient test cases could be generated using the technique described in section 8.1, if the class diagrams which are the input models could facilitate a method for including the pre and post conditions of the particular state. More case studies based on various real-life applications have to be performed to evaluate the proposals. Also how work needed in automatic test case generation could be done as a background process while the user is creating models could also be a research area. In this way the user could receive about impossible paths and transitions in the model.
14. References


http://Planet-Mde.org/, Last Accessed 27.05.2009


[18] Markus Scheidgen, “Model Patterns For Model Transformations In Model Driven Development”

[19] Margarida Afonso, Regis Vogel, Jose Teixeira, “From Code Centric To Model Centric Software Engineering: Practical Case Study Of MDD Infusion In A Systems Integration Company”


[34] Henry Muccini, “Model-Based Testing Principles”
[37] Alexander Pretschner1 And Jan Philipp, “Methodological Issues In Model-Based Testing”
[41] An Evaluation On Model Based Testing At Ericsson, Master Thesis By Jenny Florin
[46] http://www.mail-archive.com/topcased-mm-users@lists.gforge.enseeiht.fr/msg00871.html last acessed on 27.05.2009
[50] Jim Steel And Michael Lawley, “Model-based test driven development of the tefkat model-transformation engine”
[52] Amit Paradkar, “Plannable Test Selection Criteria for FSMs Extracted From Operational Specifications”
[54] Bernhard Rumpe, “Model-based Testing of Object-Oriented Systems ”
[55] Philip Samuel, R ajib Mall, Pratyush Kanth, “Automatic test case generation from UML communication diagrams”
[56] Alexandre David, Paul Pettersson, “ Uppaal Tutorial- Modeling Patterns”, RTSS’05