

Helsinki University of Technology
Department of Electrical and Communications Engineering
Networking Laboratory

Mikko Aleksi Mäkinen

Model Based Approach to Software Testing

Master's Thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in Technology

May 22, 2007

Mikko Mäkinen

Supervisor: Professor Heikki Hämmäinen

Instructor: Antti Heimola, M. Sc.

Author:	Mikko Aleksii Mäkinen	
Name of the Thesis:	Model Based Approach to Software Testing	
Date:	May 22, 2007	Number of pages: vii + 61
Department:	Department of Electrical and Communications Engineering	Professorship: S-38
Supervisor:	Professor Heikki Hämäläinen	
Instructor:	Antti Heimola, M. Sc.	
<p>Software testing is becoming more and more difficult task every day because in the current software engineering cycle the design and testing activities are separated, which leads to a situation where test cases are not in harmony with the actual application. One way to solve this problem is to take a model of the application into use, which can be, e.g., a user interface model and from which the test cases can be derived automatically. This technique is known as model based testing. The model can also be used for, e.g., automatic code generation.</p> <p>The main objectives of this thesis were to find a suitable way to deploy the model based approach into an organization, to develop a model based testing tool in order to test the approach and to show proof of financial benefits of the approach.</p> <p>The thesis is divided into two parts. First, in the literature part, the model based approach is explored thoroughly. In the second part a suggestion for the deployment of model based approach into an organization is presented, a model based testing tool implementation based on that suggestion is reviewed and a comparison between traditional testing methods and model based testing is conducted.</p> <p>The comparison between the testing methods showed that model based testing brings financial benefits to the organization. By using model based testing, a lot of time was saved in the test execution compared to the traditional testing methods.</p>		
Keywords:	model, software, engineering, testing	

Tekijä:	Mikko Aleksi Mäkinen	
Työn nimi:	Mallipohjainen lähestymistapa ohjelmistotestaukseen	
Päivämäärä:	22.5.2007	Sivuja: vii + 61
Osasto:	Sähkö- ja tietoliikenne- tekniikan osasto	Professuuri: S-38
Valvoja:	Professori Heikki Hämmäinen	
Ohjaaja:	Antti Heimola, M. Sc.	
<p>Ohjelmistotestaus vaikeutuu päivä päivältä koska ohjelmistojen suunnittelu- ja testausvaiheet ovat selkeästi toisistaan erillisiä toimintoja, jonka johdosta testitapaukset eivät vastaa ohjelmiston oikeaa toiminnallisuutta. Eräs tapa ratkaista tämä ongelma on ottaa käyttöön malli joka kuvaa ohjelmiston toimintaa esimerkiksi käyttöliittymätasolla. Mallista voidaan mallipohjaisella testauksella tuottaa automaattisesti tarvittavat testitapaukset. Mallista voidaan tuottaa automaattisesti myös esimerkiksi lähdekoodia.</p> <p>Tämän työn tavoitteina oli löytää tapa, jolla mallipohjaisen lähestymistavan voisi tuoda organisaation käyttöön helposti, kehittää mallipohjainen testaustyökalu, jotta lähestymistapaa voitaisiin testata sekä todistaa että lähestymistapa tarjoaa taloudellisia hyötyjä.</p> <p>Työ on jaettu kahteen osaan. Ensin, kirjallisuusosassa, mallipohjainen lähestymistapa käydään läpi. Toisessa osassa esitellään ehdotus lähestymistavan tuomiseksi organisaation, käydään läpi ehdotukseen perustuvan mallipohjaisen testaustyökalun implementaatio sekä verrataan mallipohjaista testausta perinteisiin testausmenetelmiin.</p> <p>Tehty vertailu todisti että mallipohjainen testaus tarjoaa suuria taloudellisia hyötyjä. Tekniikkaa käyttämällä säästettiin paljon testaukseen käytettyjä miestyötunteja.</p>		
Avainsanat: malli, ohjelmisto, tuotanto, testaus		

Acknowledgements

This master's thesis was done for Nokia Corporation between December 2006 and May 2007, and I want to thank the company for this opportunity.

My supervisor Professor Heikki Hämmäinen gave me an excellent kick-start for the writing by making me think the structure and contents of the work for which I am very grateful. The work was instructed by M. Sc. Antti Heimola whose interesting ideas and feedback also deserve thanks. Very special thanks goes to Pekka Kauppila, who struggled with the bugs in my tool and managed to actually use it to perform the tests presented later on in this work. My mate Kari Pietikäinen and co-worker Vesa Raiskila helped with refining my English and academic writing.

The work could not have been done without the support of my family and friends, and especially not without my future wife Heli.

May 22, 2007

Mikko Mäkinen

ACKNOWLEDGEMENTS	III
ABBREVIATIONS	VI
1. INTRODUCTION	1
1.1 OBJECTIVES.....	2
1.2 SCOPE.....	3
1.3 METHODS	3
1.4 STRUCTURE	3
2. MODEL BASED SOFTWARE ENGINEERING	5
2.1 PROBLEMS WITH THE CURRENT STATE OF SOFTWARE ENGINEERING.....	5
2.2 GENERAL DESCRIPTION OF A SOFTWARE MODEL.....	6
2.3 BASIC MODEL TYPES	7
2.3.1 <i>Finite State Machines</i>	8
2.3.2 <i>Markov Chains</i>	9
2.3.3 <i>State charts</i>	9
2.3.4 <i>Grammars</i>	9
2.3.5 <i>The Unified Modelling Language</i>	9
2.4 DIFFERENT APPROACHES TO SOFTWARE MODELLING.....	11
2.4.1 <i>Use Case Model</i>	11
2.4.2 <i>Application Model</i>	13
2.5 CREATING THE MODEL.....	14
2.6 MODEL BASED TESTING.....	16
2.6.1 <i>Benefits of Model Based Testing</i>	17
2.6.2 <i>Difficulties and Drawbacks of Model Based Testing</i>	19
2.6.3 <i>Currently Available Model Based Testing Tools</i>	19
3. DEPLOYMENT OF MODEL BASED APPROACH INTO AN ORGANIZATION	22
3.1 PROPOSITION FOR A STARTING POINT	22
3.2 THE FUTURE.....	24
4. DESIGN AND IMPLEMENTATION OF THE MODEL BASED TESTING TOOL.....	26
4.1 REASONS FOR IMPLEMENTING OWN TOOL	26
4.2 INTRODUCTION TO TEST AUTOMATION SYSTEM	27
4.3 PREVIOUS WORK.....	28
4.4 HIGH-LEVEL DESIGN	29
4.4.1 <i>The Model</i>	31
4.4.2 <i>File Input and Output</i>	35
4.4.3 <i>Test Case Generation and Execution</i>	36
4.4.4 <i>Graphical User Interface</i>	37
4.5 THE IMPLEMENTATION: KENDO	38
4.5.1 <i>Test Case Generation Techniques</i>	39
4.5.2 <i>Graphical User Interface</i>	44
4.6 FUTURE IMPROVEMENT SUGGESTIONS	45
5. ANALYSIS.....	47
5.1 COMPARISON TO PREVIOUS WORK.....	47
5.1.1 <i>Differences to the Previous Work</i>	47
5.1.2 <i>Advantages of the Current Tool Implementation</i>	48
5.2 COMPARISON BETWEEN TESTING METHODS	49
5.2.1 <i>Used Test Metrics</i>	49
5.2.2 <i>Test Environment</i>	50
5.2.3 <i>Results</i>	53
6. CONCLUSIONS.....	56

6.1 RESULTS.....	56
6.2 ASSESSMENT OF RESULTS	56
6.3 EXPLOITATION OF RESULTS.....	57
6.4 FUTURE RESEARCH	58
REFERENCES	59

Abbreviations

ASTE	Automatic System Test Engine
CPP	Chinese postman problem
DCG	Directed cyclic graph
DFS	Depth-first search
FSM	Finite state machine
GPS	Global positioning system
GUI	Graphical user interface
HTML	Hypertext Mark-up Language
ISO	International Organization for Standardization
JVM	Java virtual machine
MMS	Multimedia message
MVC	Model-view-controller design pattern
RTF	Rich text format
S60	Series 60 smart phone user interface
TSP	Traveling salesman problem
UI	User interface
UML	Unified modelling language

XML Extended mark-up language

XTND XML transitional network

1. Introduction

Smart phones, e.g. Nokia devices equipped with Series 60 (S60) user interface (UI), are becoming more and more complex devices containing tens of different applications and hardware add-ons that have little to do with the actual telephone part of the phone, such as cameras and GPS-devices.

Increased complexity of smart phones also leads to increased number of errors. Applications should be as thoroughly tested as possible, and the best way to achieve this is to use test automation. Designing automated test cases for the smart phones is not a simple matter. Traditionally, there are three different views on how an application should work:

- User interface designer writes a broad specification of how the user interface should look, what are the functions of different keys in different states of the application, and how the application responds to user actions.
- Developer reads the specification, and writes the application according to his/her view of the document.
- Tester takes the application and the specification and tries to figure out how the program should work and does it work that way.

These different views easily lead into troubles, since they seldom are consistent. Applications mature over time and get more features. Adding a feature to application or changing a feature means that all three views of the application must be updated, and this means that most time in testing is spend adjusting the test cases into always changing applications.

If the different views could somehow be connected to each other, testing would become a simpler task; test cases would always be in harmony with the actual application. This situation would have two great advantages:

- Applications would be more thoroughly tested because of the synchronization between the actual application and test cases.
- Costs of software development would decrease, since fewer testers would be needed to do the same job.

1.1 Objectives

A model is a way of describing how user actions and system states in an application relate to each other. If the model is thorough enough, describing every user action and corresponding system response, it should be possible to automate the test case creation and not just the test execution. This technique is known as model based testing.

In order to deploy model based testing successfully into an organization the following issues must be solved:

- A way is needed to bring model based testing to an organization “softly”, so that the initial efforts required for deployment are not too high.
- A model based testing tool that creates test cases automatically from the models must be available to be able to exploit the model based testing in an organization.
- A clear proof of financial benefits must be shown in order to make things happen in the organization.

The objective of this thesis is to find a solution to all of the above issues.

1.2 Scope

The scope of this thesis is in the user interface level of applications, although some suggestions for the future are made regarding the other levels, too. The model based testing tool is developed with S60 UI testing in mind, but it should not be too hard to modify the tool to be used with other UIs as well. S60 applications are used when comparing model based testing to traditional methods in order to show the financial benefits of model based testing.

1.3 Methods

A literature study on model based testing and model based software development will be conducted in order to obtain solid background knowledge for development of the model based testing tool.

The model based testing tool will be implemented using Java based on the results of the literature study. This allows actual usage of model based testing, since commercially there is not a single suitable tool available. The implementation also serves the purpose of learning; it is possible to see how the model based testing should be used and are there any advantages over traditional testing methods.

In addition, a comparison of traditional test cases and test cases produced by the tool implemented will be done. The comparison will be made using S60 Image Viewer and Voice Recorder applications as a reference point. Code will be instrumented, and the instrumentation figures obtained by using both testing methods will be compared. The result of the comparison will ultimately decide whether it is worthwhile to continue on the model based testing track.

1.4 Structure

Chapter 2 consists of a literature study of model based testing and model based software development.

Chapter 3 contains suggestions how to deploy the model based approach into an organization. The chapter mainly involves discussion about a good starting point on a road to model based software engineering, but some future suggestions are presented also.

In Chapter 4, an implementation of model based testing tool will be presented. Since the implementation depends heavily of the test system used, an introduction to it will be presented also in the chapter. The actual code or any coding related details will not be presented; implementation will be described in high level although some relevant algorithms are described.

Chapter 5 contains the comparison between traditional test case execution and model based testing.

Conclusions of the results obtained in this work and future research suggestions are discussed in Chapter 6.

2. Model Based Software Engineering

In this chapter, a literature study of model based software engineering is conducted. First, the problems of current software development processes and reasons why model based approach would be better are examined. After that, model in general and different types of models and their applicability to different types of software modelling are presented. The last two sections contain an overview of model based software development and testing.

2.1 Problems with the Current State of Software Engineering

In general, current process of software engineering is best described with the popular V-model [1] (Figure 1). The V-model describes the whole software engineering process, starting from the left tail's specification and design activities, going through coding stage, and ending up in the right tails testing activities.

As can be seen from the figure, design and testing are connected through the planning phase, but are still clearly two separate actions. The planning phase traditionally consists of a tester studying the different parts of the software system and then writing and executing individual test scenarios that exercise the system [2].

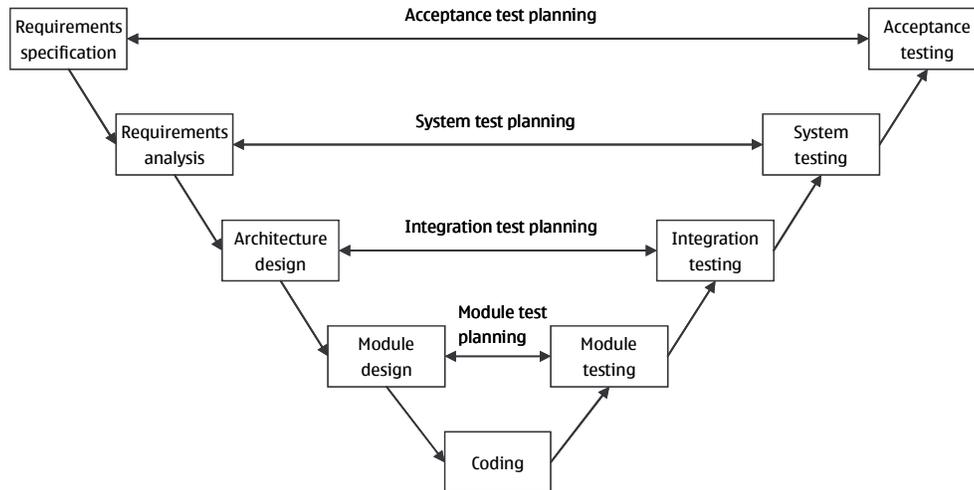


Figure 1: The V-model

This approach to testing has at least two major challenges [2]:

- The traditional tests will suffer from the “pesticide paradox” [3], in which the bug fixing leads to less and less efficient test cases since the bugs they reveal have already been fixed.
- As software tends to change over time as new features are added, the tests must be modified to support those changes. Modification of the tests takes time and since time is money, the maintenance costs can rise high if modifications are required often.

One solution to those problems is to take a model of the software into use. With the model based approach, there is no need for separate test planning or even coding as the test cases and code can be generated automatically from the model.

2.2 General Description of a Software Model

In order to explain the model based approach fully, a definition or a general description of a software model has to be found. In [4], it is defined as following:

“Simply put, a model of software is a depiction of its behaviour. Behaviour can be described in terms of the input sequences accepted by the system, the actions, conditions, and output logic, or the flow of data through the application’s modules and routines. In order for a model to be useful for group of testers and for multiple testing tasks, it needs to be taken out of the mind of those who understand what the software is supposed to accomplish and written down in an easily understandable form. It is also generally preferable that a model be as formal as it is practical. With these properties, the model becomes shareable, reusable, precise description of the system under test.”

Models can help to predict behaviour of software since generally they are simpler than the actual software they describe [5]. The prediction of behaviour is essential for the automated test case creation and thus for model based testing. One important, but often neglected, requirement of software models is that the model must be more abstract than the software it is used to model [6]. This requirement is because that in order to have any benefits from the modelling the model must be easier to validate than the actual software. If the model is at the same abstraction level as the software, the efforts of validating that the model is correct are the same as validating that the software is correct.

Software can be modelled in a number of ways, and the most important ones are introduced in the following section.

2.3 Basic Model Types

Software can be modelled in numerous different ways, each of those having its own pros and cons. The method of modelling has to be selected to suit the software that is to be modelled. In [4] the most common ways to model software are presented, and those are covered next.

2.3.1 Finite State Machines

Finite state machine (FSM) is a behaviour model, which consists of states and transitions between those states. An example FSM is visualized in Figure 2.

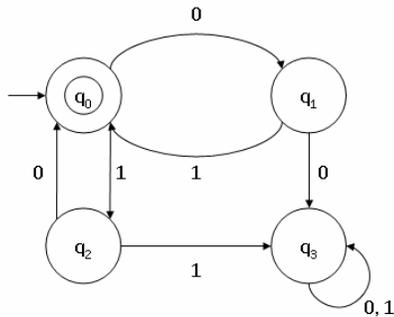


Figure 2: Example finite state machine

The example FSM is a simple binary machine, where q_0 is the start state, and q_3 the final state. Transitions between the states occur either with binary 1 or with binary 0. For example, sequence 00 would go through states q_0 , q_1 and end to q_3 , and sequence 10101 through q_0 , q_2 , q_0 , q_2 , q_0 , and end to q_3 .

FSM can be described formally as quintuple (I, S, T, F, L) , where

- I is the set of inputs of the model
- S is the set of all states in the model
- T is a function which determines whether a transition occurs when an input is applied to the system in a particular state
- F is the set of final states where the model can end up
- L is the state where the software is launched

The states and the transitions can be used to visualize almost any interaction between any two systems, e.g., user and a coffee machine or two computers communicating with each other with some protocol.

2.3.2 Markov Chains

Markov chains are stochastic models, which can also be used to software modelling. It could be said that they are probabilistic state machines, meaning that the transitions of the machine contain a probability, which is used to select which transition to choose whenever leaving a state. They can be easily used to measure, e.g., software's reliability and mean time to failure.

2.3.3 State charts

State charts are an extension of finite state machines. They are used to model complex or real-time systems, both of which are not possible to do easily with finite state machines. In state charts, it is possible to specify state machines in a hierarchy, where upper level states contain a complete lower level state machine, or several machines. It is also possible to add certain external conditions into the model that determine whether certain transitions are possible or not.

State charts are an equivalent to the Turing machine. They are easier to read than finite state machines, but nontrivial to work with and require training beforehand.

2.3.4 Grammars

Grammars are used to describe the syntax of programming and other input languages, and they are equivalent to different forms of finite state machines, so they can be used e.g. to represent a model of a system in a more compact form than finite state machines do. They are also easy to write and maintain.

On the other hand, grammars do not provide, as such, any visualization of the language or model they represent.

2.3.5 The Unified Modelling Language

The unified modelling language (UML) is similar to finite state machines, except for the fact that it is used to describe complicated behaviours of software, thus the simple graphical representation of a state machine is replaced by a structured language.

UML contains two parts: a model, and a set of diagrams. The model could be best described as a formal description of the diagrams, which are used for visualization, and it contains three essential parts:

- Functional model, which shows the system functionality from the user's point of view
- Object model, which contains the system structure (objects, their methods and attributes and relationships between them)
- Dynamic model, which shows the internal behaviour of the system using e.g. sequence, activity and state machine diagrams

UML diagrams are divided also to three parts:

- Structure diagrams describe all the components the modelled system contains; e.g. classes, packages, objects and the overall structure of the system
- Behaviour diagrams are used to describe how the system actually works. This includes e.g. workflows and use cases.
- Interaction diagrams describe the data and control flows of the system

If the diagrams are explored more thoroughly and deeper than what was presented, it becomes apparent that the UML may contain more or less every model type described earlier in this chapter at least in some form. This leads to increased complexity; UML can be used, and is in fact designed to be used for modelling complicated software systems at every level possible, from the low-level class diagrams and e.g. protocol state machines to high-level user interactions, but due to its complexity it is not, as a whole, suitable for simpler modelling tasks. Parts of UML, e.g., use case diagrams, can be efficiently used for simple tasks, but then question arises whether the modelling is done with UML or with something else.

2.4 Different Approaches to Software Modelling

Because of the scope of this thesis, the focus of this section, and the rest of this thesis, is only on modelling software on the user interface level.

UI works by user executing different actions and the system (UI) going into different states because of those actions, and because of that, it is clear that finite state machines are the obvious choice for modelling software at the UI-level. This is because of the visual simplicity they provide¹, in contrast to UML, which does not suit well for modelling systems as simple as UI.

The finite state machine contains a set of states, user actions and system responses, and their relations to each other [7]. User actions and system responses are called events from now on in this thesis. The finite state machine can be used e.g. to describe a use case, or the whole application. The differences between these two approaches are presented next.

2.4.1 Use Case Model

Use case is a description of one simple task that a user performs, e.g., taking a picture with the camera, from the users point of view. In other words, use cases are used for capturing user requirements without excessive technical jargon, which is usually the case with software design documents.

Use case model consists of several user events and their corresponding responses, the system events. After combining the events together, they define the whole use case. The combination is called the flow of events. One way of defining use cases is UML and its use case diagram. An example diagram representing a simple restaurant is in Figure 3. As can be seen, the purpose of the UML use case diagram is only to visualize the different actors of the system and their possible actions.

¹ One of UML's behaviour diagrams uses finite state machines for model visualization, but in author's opinion, it makes more sense to talk about finite state machines rather than "UML's behaviour diagrams."

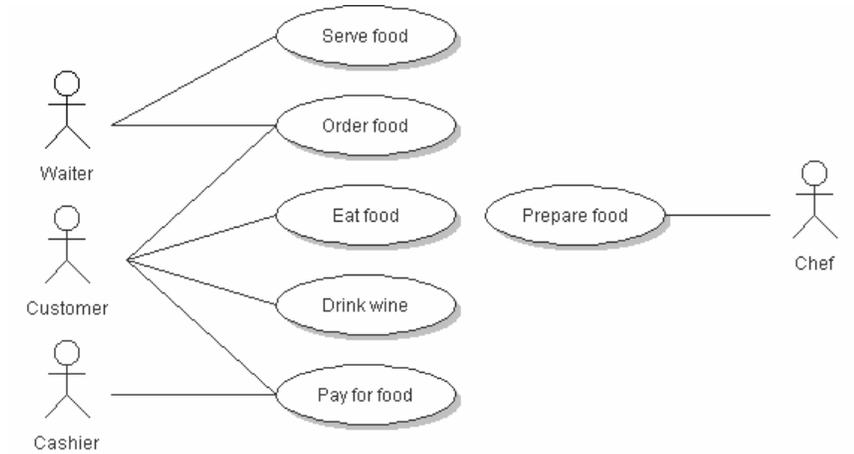


Figure 3: Example of UML use case diagram

1.1.Capture Image

1.1.1.Description
This use case defines the steps for capturing image in S60 camera application

1.1.2.Pre-condition
Camera application is open

1.1.3.Post-condition
None

1.1.4.Flow of Events

	User		System
U.1	Press CenterPush key to capture image [S.1]	S.1	Image is captured. Text §Image§ is displayed [U.2.x]
U.2.1	Press Clear key to delete captured image [S.2]	S.2	Delete image confirmation is displayed with text §Delete?§. [U.3]
U.2.2	Press SoftKey §Options§ and Select Menu §Delete§ to delete captured image [S.2]		
U.3	Press SoftKey §Yes§ to delete image. [S.3]	S.3	Image is deleted and Camera application returns to initial view.

Figure 4: Example use case document

In order for a use case to be usable, it should be defined more formally than the UML's use case diagram is, to ensure that everyone who reads it understands it exactly the same way. In [8] a formal template of a use case document was presented. The template is originally defined in [9]. Use case documents contain general

description about the use case, pre- and post-conditions (system states) for the use case and a use case event table, which is a formal representation of the flow of events. A simple example of use case document describing how to capture an image using S60 camera application is in Figure 4.

Use cases are a good way to describe the different requirements the users have for the system and it is also proven that automated test cases can be derived from the use cases easily. On the other hand, since the use cases of one system do not interconnect in any way, it is possible and even probable that they do not model the system as thoroughly as possible. An attempt to increase the model's coverage of the system is to use so called application model.

2.4.2 Application Model

The application model, in contrast to use case model, describes the behaviour of a whole application, e.g. a music player or a clock. Speaking in terms of UML, the application model is the same as behaviour model. The model contains every entry and exit point of an application and every state possible between them. Thus, every user action is also modelled.

As an example, a simple application model of the Windows Sound Recorder is presented in Figure 5. The model contains only two states, stopped and recording. Transitions to different states occur when user clicks on either the record-button or the stop-button.

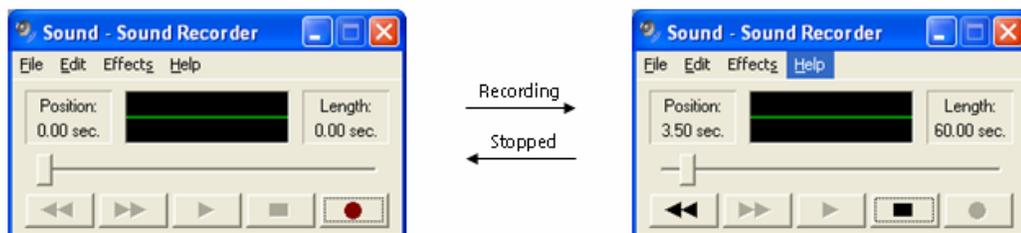


Figure 5: A simple application model of the Sound Recorder

As with use case models, also application models should be based on formalized documents, describing the states of the application, user actions between them and the

general flow in a sensible manner. Usually some kind of documents are created by user interface (UI) designers to describe the UI, but in general the documents do not contain as clear relationships between the states as is needed. In addition, as UI documents tend to be overly verbose they are not feasible to be used with test automation. The use case document template presented in the previous section can also be used to describe the whole application with little or no modifications.

2.5 Creating the Model

A fact is that software models do not come out of thin air; they must be created somehow. This fact is often neglected in research papers on model based testing.

In [4], the following three tasks are suggested for software modelling:

- Understand the system under test by learning about the software and its environment. Learning includes e.g. determining the components that need to be tested, reviewing available documentation, communicating with other teams involved in the software engineering, potential user identification and documenting every input, response and the sequences of inputs that need to be modelled. This also possibly includes the maintenance of all gathered information, i.e. the model.
- Choose the right model type for the software being modelled. There are also other factors for choosing the model than that the model is adequate for the software; e.g. people who are going to actually work with the model have to be taken into consideration.
- Build the model by making a list of all possible inputs to the system, document the situations where user can and cannot actually apply the input and document the situations where input causes different responses from the system.

After the model is built, it has to be transferred into state machine. Several algorithms for automating this task exist; some are presented e.g. in [10].

Harry Robinson suggests for model creation an approach called operational modes in [5]. Operational mode describes what user actions are possible in a state and what the system responses are when the actions are executed. Although the use of operational modes is suggested because “a finite state model used in representing the behaviour of an application is likely to have many, many states – so many that it would be tedious and unrealistic to create and maintain the model by hand”, no solution is presented how the actual operational modes should be created and maintained so that the task would not be tedious and unrealistic.

Both approaches to model creation discussed above are probably good if it is wanted that the model be created manually. Although algorithms for creating state machines from either one of discussed approaches exist, a lot of manual work is still required before it is possible to use any automation to aid the creation of the state machines.

One solution for reducing the manual creation work is to use the model throughout the software engineering process, starting from the design. If the software is described as a model from the start, no separate creation process is needed. This approach is commonly known as model driven development.

The defining characteristic of model driven development is that the primary focus and products of software development are models rather than computer programs [11]. The major advantage of this approach is that models are expressed using concepts that are less bound to the underlying implementation technology and closer to the actual problem domain. Modelling can be done in any language or with any method; e.g., UML can be used as well as UI specifications. The choice of the modelling language depends on the level of modelling wanted.

The basic idea of model driven development is to automate the transformation of models from one form to another [12], which means that models are imported from some form which could be, e.g., the methods described before and exported to another form; e.g., C source code. In order for the transformation to work, the import and export languages have to be defined precisely; the language definition can be chosen freely, which means that the models can be transformed practically from anything to anything. The key factor in transformation is the mapping of source language to the destination language. As an example, UML class diagrams can be

easily mapped to Java class skeletons, and UI specifications to test automation scripts.

Models are not so useful if they end up only as documentation that is used as a reference during the software development; the maximum value of modelling is only achieved if models are used as seeds for automatic code generation and verification activities. The verification part is known as model based testing, and it will be covered later on in this chapter.

The ultimate vision of automatic code generation is that the model would take the role of implementation languages, analogous to the way the third-generation programming languages (e.g. C++) displaced assembly languages [11]. Code generation would not produce only code skeletons, but complete and working programs. There would be no need to examine or modify the generated programs, just as there is no need to examine or modify the assembly language produced by third-generation programming language compilers. This, of course, is a goal that is hard to achieve, but as it is said in [11], there already exists tools and techniques that generate better code than handcrafted would ever be, even when the models describe large-scale industrial applications.

2.6 Model Based Testing

Model based testing can be summarized in one sentence; it is essentially a technique for automatic creation of test cases from specified software model. The 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 [13].

Generating test cases from a finite state machine is usually straightforward and easy task. The states in the machine contain information about what the state is; in the case of actions, they describe the action that is executed, and in the case of responses, they describe formally, what the response is. If the test cases are to be executed manually, this is all the information that is needed. If the goal is to generate automated test cases, the states also need to contain all the necessary test automation parameters that

are needed by the test automation system used. The test automation parameters can of course be derived also from some other information, e.g. by using a text parser to generate the parameters from the verbal descriptions automatically.

As the states contain all the necessary information, actual test case generation is simple. The simplest approach is to traverse randomly through the state transition diagram, from start to the end. One test case is constituted of the states visited during traversing. Frequently it is wanted that the transition coverage criterion is 100% and then random traversing is out of the question and a more sophisticated algorithm is needed. In [2], some graph traversal algorithms are presented that aim to meet the criterion, and the challenge is to choose the most effective one.

In [4] several advantages and disadvantages of model based testing are presented, and those are explored next.

2.6.1 Benefits of Model Based Testing

Many studies conducted have shown that model based testing is effective, especially when used to test small applications, embedded systems, user interfaces and state-rich systems with reasonably complex data. Rosaria and Robinson (2000) studied testing graphical user interfaces, Agrawal and Whittaker (1993) embedded control software and Avritzer and Larson (1993) phone systems.

Usually the most attractive attribute of model based testing is thought to be the automatic generation of test cases, but that is not all. Model of software can help refining unclear and poorly defined requirements [13]. By eliminating model defects before the coding begins and automating the test case creation the result is significant cost savings and higher quality code. Figure 6 [14] shows the differences between current defect discovery and elimination process (marked “Old”) and early defect discovery (marked “New”).

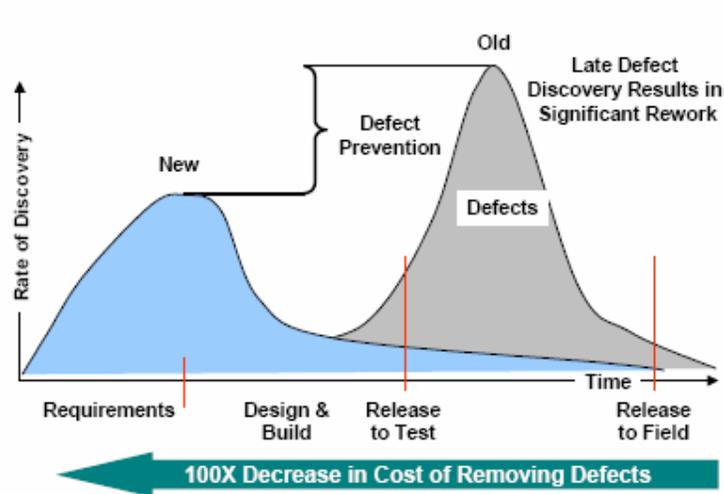


Figure 6: Savings caused by earlier defect discovery

The figure shows that the sooner the defects are detected and fixed, the less the costs of fixing them will be.

Other benefits that are more related to testing include e.g. the following, which were presented in [13]:

- Comprehensive tests; if the model is a complete abstraction of the software, it is possible to automatically create test cases which cover every possible transition of the model by using graph algorithms.
- Defect discovery; model based test automation discovers defects more effectively than manual methods. The article demonstrated this with a case study in which manual method uncovered 33 defects in a system, and model based method all of those and in addition 56 more.

As it was shown in this section the benefits of model based testing are huge if modelling and all the related tasks are done efficiently, but it also has some difficulties and drawbacks.

2.6.2 Difficulties and Drawbacks of Model Based Testing

Almost every research on model based testing agrees on one thing: deployment of model based testing into an organization requires significant efforts and investments. In [4], the following three reasons for the needed efforts and investments are presented:

- Excessive amount of skills is required from the testers. They need to be familiar with the model, which means knowledge of different forms of state machines, formal languages, and automata theory. In addition, expertise in tools and scripts is required when test automation is going to be used.
- A large initial effort in terms of man-hours is required. The type of the model has to be carefully selected, different parts of software have to be divided so that the modelling is easier because of the smaller areas and the actual model has to be built.
- Models themselves have also some drawbacks. The biggest one of those is the explosion of state-space needed. Even a simple application can contain so many states that the maintenance of the model becomes difficult and tedious task.

As can be seen from the list, model based approach to software testing is not the perfect solution. The positive side though is that all the points in the list can be overcome with thorough planning of the deployment of model based testing into an organization.

2.6.3 Currently Available Model Based Testing Tools

A working model based testing tool is a requirement for deploying the model based testing into an organization, and this section contains a small study of currently available tools. Most of the tools in the list below were found from [15], and some simply by surfing the net and reading articles on the work done on model based approach.

The tools listed are either commercially or otherwise publicly available.

AETG[™] by Telecordia[™] Technologies is a web-based pair wise test case generator, which uses the input data domain as the model. This means that the tools target use is with e.g. databases or user input data validation.

Case Maker[®] by Diaz & Hilterscheid Unternehmensberatung GmbH is also a pair wise test case generator, but in addition to using the input data domain as the model, it uses also constraints specified by business rules, i.e. the business domain.

Conformiq Test Generator[™] by Conformiq generates test cases from UML state charts, which represent a high-level graphical test script. It has to be underlined that the state charts do not represent the actual system under test but only the test script, which means that the tool is more a test script editor than a real model based testing tool.

CTesK, JTesK by UniTESK uses formal representations of software requirements as the basis of test case generation. The requirements are written directly to the source code with the help of extensions to the programming language, and the process of transforming the requirements into test cases is automated.

MaTeLo by All4Tec generates test cases from usage model of the system under test. The model is created by hand with the editor included, and is based on Markov chain theories.

Qtronic[™] by Conformiq automates the test case generation from software design models. The Conformiq web site says that it can import UML models from several tools, and analyze and verify the models. This sounds promising, but when tested, the tool did not support e.g. model import. In addition, the modeller tool that was included in the package did not do anything sensible.

Rave[™] by T-VEC is used to model and analyze system requirements before the coding begins. The requirements models are then used to generate test cases automatically. The modelling is done with so-called tabular notation editor that is also available from T-VEC.

Reactis[®] by Reactive System is targeted for model based testing of embedded systems, and it generates the tests from Simulink[®] and Stateflow[®] models.

SmartTest by Smartware Technologies uses the same pair wise technologies as AETG and Case Maker do, and the model is the input data domain.

Statemate™ Automatic Test Generator™ / Rhapsody® Automatic Test Generator™ (ATG) by I-Logix generates test cases from state charts of the system and from UML state machine. The state charts have to be designed with the included designer tool.

TAU Tester by Telelogic uses TTCN-3 as the modelling notation and generates tests from it automatically. The models have to be created by hand.

Test Cover by Testcover.com is a web-based service, which, like AETG, Case Maker, and SmartTest, uses pair wise techniques and uses the input data domain as the model.

TEMA Tool by Research Group on Software Engineering, Institute of Software Systems in Tampere University of Technology is currently used e.g. for S60 UI testing. It is based on the use of test models, which specify the behaviour of the system under test from the perspective of testing. The tool is a test case generation engine and it utilizes commercial tools used in the industry by extending them for the purposes of executing test cases generated from test models.

T-Vec Tester for Simulink® - T-Vec Tester for MATRIXx® by T-Vec is used mainly for testing of embedded systems. It generates test drivers and test vectors from either Simulink or MATRIXx models.

ZigmaTEST Tools by ATS uses a finite state machine based models, from which it can generate test sequences to cover the state machine transitions. The finite state machines must be created by hand.

3. Deployment of Model Based Approach Into an Organization

One of the key problems in bringing model based testing into any organization is that the model needs to be created and be involved in the whole software engineering process, not only in testing. Several researches, e.g. [4, 5, 13], suggest that testers would be the ones to create the models used in testing, but as said in section 2.6.2, this approach would require too much skill from the testers and thus is not applicable.

In section 2.5, model based software development was explained. Although the prospects of it, e.g. automatic code generation, are tempting, it is not possible to convert a real organization's processes from traditional software development to model based software development in a short time period. A simple, yet powerful, starting point is needed on which the transition from traditional ways to model based software engineering can be built on one small piece at a time. The starting point must be easy to take into real use, and on the other hand it must provide at least some clear cost and man-power benefits to the organization.

3.1 Proposition for a Starting Point

One possible starting point would be to utilize the work of user interface designers. When designing the UI, they actually describe how the application flow goes in the UI level. As was said in section 2.5, the basic idea of model driven development is that the models act as transformers between different sources and targets. Source can be a UI specification, if the document is in a formal form, and target e.g. a test automation script. Figure 7 contains a modified V-model, which visualizes how this transformation fits in to current software engineering process. The model replaces the system test planning phase completely by automating the process.

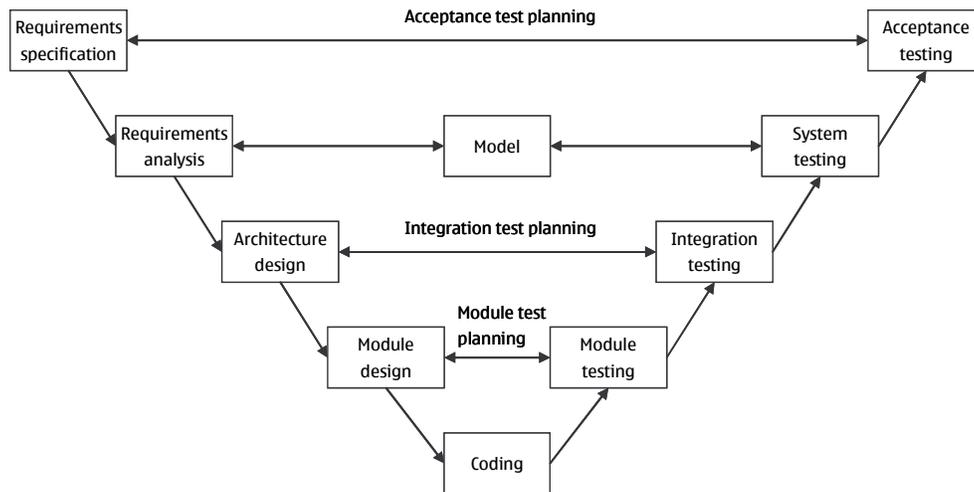


Figure 7: Modified V-model

The presented approach should not require any extra work from anybody, since the already existing work of UI designers is exploited in a new way. The exploitation requires that a tool is available which can convert UI specifications into an application model, and create test cases from it.

Once the model has been created automatically from the design, the testers only have to add the test automation parameters into the model. Of course, they also have to execute the tests and analyze the results but that is already a part of their job. The adding of test automation parameters is only a one-time job so it should not be any extra burden for the testers. It is also possible that the UI specifications would be written in such a way that the test automation parameter adding could also be automated by using a text parser, but this would require a formal way of creating the designs.

One possibility to write the UI specifications in a formal form could be to make use of the use case document template presented in section 2.4.1. The flow of events table in the template can be used to describe the whole UI, since it contains user actions and the corresponding system states. In the case of S60 UI, which contains several small applications, the design document should be divided into several smaller pieces, consisting of the small applications, and to one higher level document, which

describes how the main UI works and what are the relationships of different small applications. For example, it is possible to enter the messaging application from various other applications such as camera or image viewer. The higher level, or main, document would describe the inputs and outputs of the small applications in a higher abstraction level, and the other design documents the inner workings of the applications.

The approach to the structure of UI specifications presented would be formal enough for the model creation, and in addition to the formalism, the levelled structure would also make the specifications easier to read since the higher level would not contain too many details about the inner workings of separate applications. If more detail is needed, a separate, small application specification can be used.

Because the proposition presented is only a starting point, discussion about the future is also needed.

3.2 The Future

In the future, software model should be not only a helper for creating UI tests, but an essential part of the whole software engineering process. This is visualized in Figure 8, in which further modifications are applied to the already modified V-model (Figure 7). The test cases derived from the models would not be only for the UI-level; test cases could be derived also to all other levels including module, integration, and acceptance testing.

If model driven development, which was presented in section 2.5, would be taken into use, the products of software development would not be source code and programs as such, but models from which the code and tests could be easily derived. In short, it would be possible, if the models were good enough, to automate every other step of the software engineering process except for the design part. This would result in tremendous savings in man-hours, since automated tools would do most of the work.

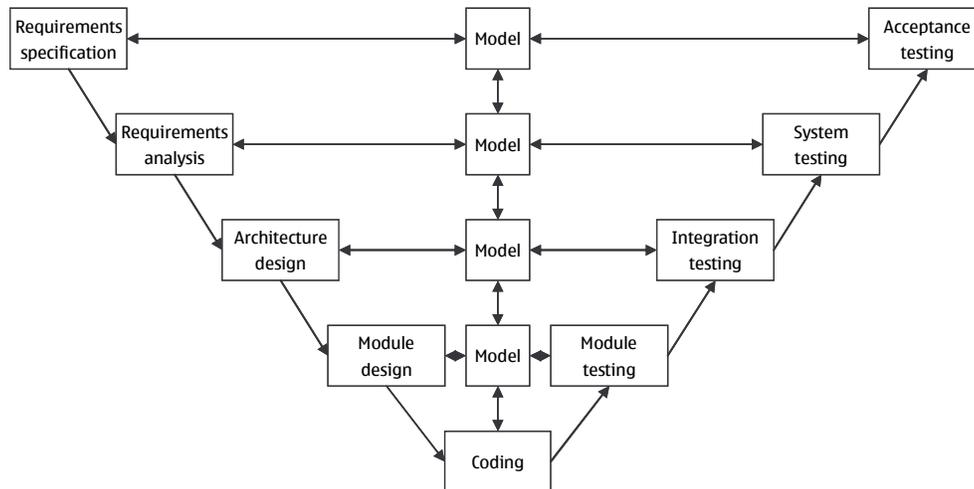


Figure 8: Further modified V-model

Of course, several questions arise when thinking about automatic code and test generation; e.g., whether the code is correct and optimized enough, and whether the code coverage that is achieved with automatic test generation is at an acceptable level are among those. These questions are of course relevant and justified, but maybe there is no reason for being worried; as it is said in [11], the code correctness and optimization questions were raised already over 40 years ago when the first compilers were introduced. The same techniques that are used currently when compiling higher-level languages into machine code can be applied to transformation of the model into high-level language, and therefore the correctness or optimization are the smallest problems the model driven development will face.

The biggest problems are related to the actual deployment of the model based approach into a big organization. The wheels of a big software development organization turn slowly and the processes are not easily changed, which bring great obstacles into the way of deployment. Also at the moment, it seems impossible to automate everything, but the change has to start from somewhere.

4. Design and Implementation of the Model Based Testing Tool

As said in the introduction, a fully functional model based testing tool is a requirement if model based approach is wanted to take into full use in an organization. One of the objectives of this thesis was to implement a production quality tool, and the objective is realized in this chapter.

First, the reasons for choosing to implement a new tool instead of using already existing one are discussed. After that, the test system, which the tool is currently using, is introduced briefly. Finally, the actual design and implementation of the tool are presented.

4.1 Reasons for Implementing Own Tool

A small study of already existing model based testing tools was made in section 2.6.3. The common problem with all of the tools studied was that they work only with already existing application models, and this leads to problems with where the models come from described in section 2.5. In section 3.1, a solution to the problem was proposed in the form of a modified V-model (Figure 7).

The first requirement for the tool comes from the modified V-model, and is that the tool has to support converting UI specifications to a model, and the model to test automation scripts. This has to also work backwards in order to make it possible to export the models created in the tool to UI specifications and test automation scripts created by hand into model skeletons, so that *the model is a two-way translator between the UI specification and test cases*, as visualized in Figure 9.

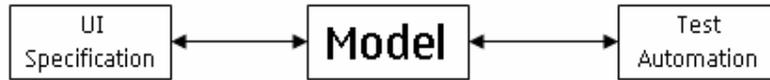


Figure 9: Model as a translator

Of course, it has to be also possible to create the model from a scratch, without the need of any kind of existing documents or designs.

Second important requirement is that the tool has to support the test case generation as described in section 2.6, i.e., randomly generated test cases and test cases for which the model coverage is 100%.

Third requirement is related to graphical user interface of the tool and especially to usability. The tool has to be so easy to use that everyone who is involved in the software engineering process can, without too much work, learn how to use it, and preferably so that they do not even have to read the user manual beforehand.

4.2 Introduction to Test Automation System

The test automation currently supported is based on Automatic System Test Engine (ASTE), which is a keyword based test automation tool developed in Nokia. ASTE reads test scripts and sends corresponding commands via a data gateway application to the device under test.

Currently ASTE can be used to test several different S60 smart phones and smart phone emulators. Test automation is based on running scripts, which consist of keywords and their parameters. Keyword could be best described as one user action, e.g. pressing a key.

There are several different logical groups of keywords, but the two interesting ones are action keywords that execute some user action and verify keywords that verify that something is displayed on the screen of the device under test. For example, action keywords can be used to type something on the screen or to select some menu, and the verify keywords to verify that some text or picture exists on the screen. There are also some performance related and miscellaneous keywords.

Traditionally test cases are created by hand as scripts for ASTE. An example test script is presented in Figure 10.

```
1 1.1 Capture Image (#00000011)
2
3 Test objectives:
4 This use case defines the steps for capturing image in S60 camera application
5
6     kw_AppState $Camera$
7     kw_PressHardkey <CenterPush>
8         vw_VerifyText $Options$
9         vw_VerifyText $Back$
10    kw_PressHardkey <Clear>
11        vw_VerifyText $Yes$
12        vw_VerifyText $No$
13    kw_PressSoftkey $Yes$
14
```

Figure 10: An example test script

The example test script looks like the following from the ASTE point of view:

- 1) Go to application state camera.
- 2) Press the centre selection key
- 3) Verify that texts “Options” and “Back” actually exist on the screen
- 4) Press the clear-key
- 5) Verify that texts “Yes” and “No” actually exist on the screen
- 6) Press the “soft key” “Yes” to exit

Depending on the results of the verification keywords, ASTE sets the test case to either passed or failed. A log is also written in HTML-format, where the failures can be seen in a user-friendly format.

4.3 Previous Work

In [8], a predecessor to the tool developed in this thesis was implemented. The predecessor was based on the use case approach described in section 2.4.1. Because

of that, the previous version and the version implemented in this thesis are fundamentally different; use cases differ radically from the application model used as the basis of this work.

Following features were implemented in the prototype:

- Use case design
- Use case import from use case documents
- Prioritization of different paths in the use case
- Test case generation and execution

As can be seen from the list, the previous version supported only use case models (section 2.4.1), and application models were not supported at all. Use case models were not allowed to contain loops, and thus the test case generation was done with a simple but appropriate depth-first search (DFS) algorithm.

Many of the features implemented to the tool during this work are based on the future improvement suggestions in [8]. The suggestions included e.g.:

- Use case export to use case documents
- Support for test data
- Multiple use case support
- UI improvements

The previous version was a good starting point for this work, but this version was designed and implemented entirely from a scratch. The design and implementation are presented next.

4.4 High-level Design

As the goal of the design was to keep the tool as modular as possible, the architecture was divided into four main blocks:

- The actual model, which is at the core
- File input and output
- Test case generation and execution
- Graphical user interface

The architecture is visualized below in Figure 11. As can be seen from the figure, the architecture is straightforward and allows high level of modularity. For example, every single part of the tool can be changed to another easily because the parts are not too deeply tied to each other. It is possible to add file inputs or outputs easily, the test generator can be changed to produce test automation scripts for a different test automation system, and the GUI can be changed to e.g. a command line interface.

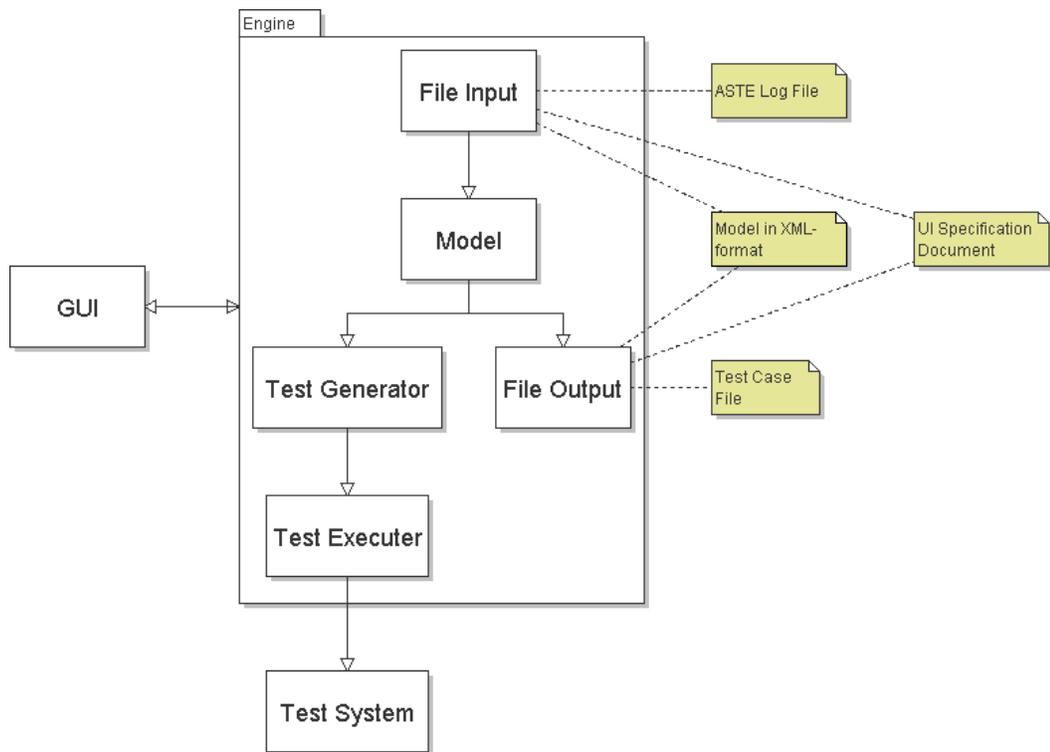


Figure 11: The high-level architecture

Since the graphical user interface (GUI) is an essential part of the tool and because modular architecture was wanted, model-view-controller (MVC) design pattern was

used to build the relationship between the actual engine (as shown in Figure 11) and the GUI:

- Model is the representation of the data that the application uses. It is possible, but not required by the design pattern, that the model also encapsulates the data access functions. The encapsulation is used in this design.
- View is typically the GUI element (as opposed to a common misconception that the view is the whole GUI) which shows the model in some, specified form. In this case, it could also be thought that the different file formats, which are outputted and inputted, and the generated test cases, are just different views of the same underlying model.
- Controller processes and responds to events and depending on those may change the model in some way. Events are usually related to GUI and user actions, but also other kinds of events are possible; e.g. a file changes on the hard disk and controller is monitoring it, this leads to modifications to the model.

The clear advantage of this design pattern is that the different components are separated so that the model is in no way dependent of the view or the controller. This allows a highly robust and modular architecture, which was also one of the goals.

Different parts of the architecture visualized in Figure 11, i.e., the model, file input and output, test case generation and execution and the GUI, are presented next.

4.4.1 The Model

The architecture of the actual model is visualized in Figure 12. As seen from the picture, the model consists of name and general description of the model, pre- and post-conditions (at what state the system must be when the execution of the model starts or stops), test data and a graph that is used as a data structure to store the actual finite state machine.

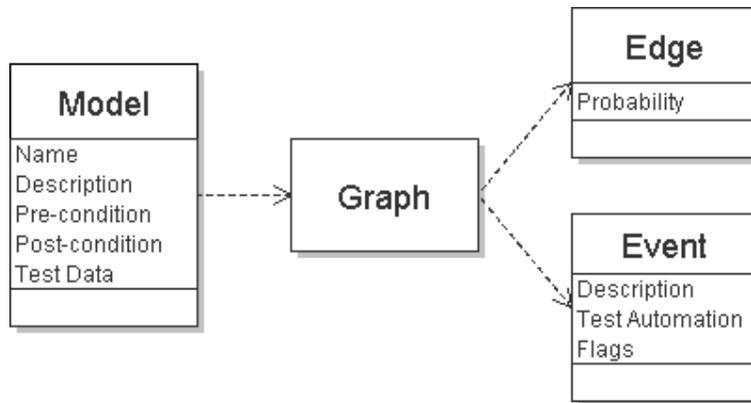


Figure 12: The model architecture

4.4.1.1 THE GRAPH

The data structure used for storing the finite state machine is directed cyclic graph (DCG), which means that every transition is traversable only into one direction and that graph may contain cycles. An example graph is in Figure 13.

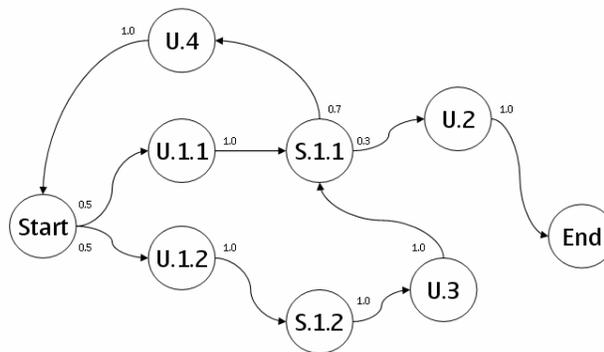


Figure 13: An example graph

Graph contains user and system events, and transitions between them. When comparing the example graph presented in Figure 13 with the example finite state machine from Figure 2, it can be seen that the graph is not a pure finite state machine, since the transitions contain probabilities, which are borrowed from Markov chains.

The probabilities are used to simulate a situation where user e.g. when in idle-state presses the menu-key 80% of the time and other keys combined 20% of the time. This allows long-term testing which simulates actual human behaviour as accurately as it is possible by mathematics.

Events contain description about what the event is and test automation keywords as described in the previous section. Because sometimes it might be wanted that events are enabled or disabled dynamically when certain conditions are met, the events also contain so-called flag variable. The flags take the graph a little more away from the pure finite state machine, since the external conditions, which enable or disable events, is a feature from the state charts. The disabling of events is visualized in Figure 14.

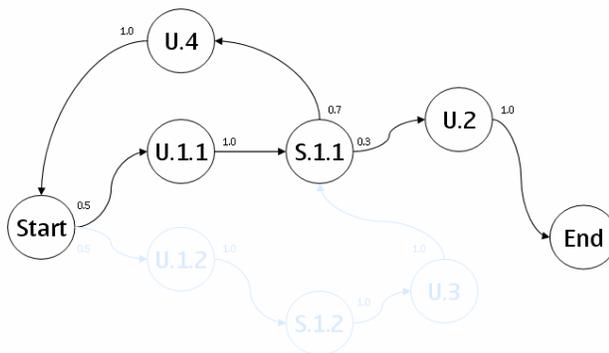


Figure 14: Events disabled

There exist two types of flags: static and dynamic. Static flags are used when a certain application feature needs to be turned off for the current test case; e.g., it is possible that it is wanted to test same model with several different devices, which either contain or do not contain a camera. By setting flags depending on the device, it is

possible not to allow creation of any test cases that use camera. One good example could be the sending of a MMS-message, where the model could contain an event where picture is taken with the camera and then sent by MMS which is clearly not possible if the device under test does not contain a camera.

Dynamic flags can be used for various tasks, but mainly their intended use is in the case of changing application settings. For example, in the S60 Voice Recorder application it is possible to change the quality of the recording to either high or low. Certain events in the model could contain some test automation that is dependent on the quality being either one. By using dynamic flags, it is possible to disable transitions to events that are dependent on e.g. high quality when the quality is changed to low at some other event.

4.4.1.2 TEST DATA

Test data is used when it is wanted that the test automation parameter contain several different options. For example, if a test case is testing that certain file is opened and correct message, “File opened successfully” or “File type is not supported!”, is shown to user, it would not make sense to manually change the parameters into desired ones if it is wanted to test a set containing several different files of correct and not correct types.

In the tool, the test data is stored as tables in the model. An example test data table is presented below.

Table 1: Example test data

Key word	Verify word
CorrectFile.doc	File opened successfully
CorrectXMLFile.xml	File type is not supported!
CorruptFile.doc	File type is not supported!

Now, instead of setting test automation parameters manually, e.g. *kw_OpenFile CorruptFile.doc* and *vw_verifyText “File type is not supported!”* the test data is added by replacing the parameters with the test data’s identification string. The tool

will automatically generate three separate test cases, as there are three separate entries in the test data table.

Of course, if more than one test data table exists, also more test cases are generated. Number of test cases is defined by the number of possible combinations of test data. For example, with one table with three entries and another table with two entries would lead to six different test cases.

4.4.2 File Input and Output

As one of the tool's main functions is to work as a mediator between the UI specification documents, models, and test cases, file input and output functions are extremely important parts of the tool.

The actual model needs to be stored to hard disk in some format. Since flexibility is a desired feature, extensible mark-up language (XML) was chosen as the format. The XML's structure is based on XML transition network definition (XTND) specification [16] with some modifications in order to enable the storage of test automation and other, general information.

It was chosen that UI specification documents can only be imported from and exported to RTF-format. The decision was made because, as an example, Microsoft Word's document format is closed, and reading in reliably would be either a difficult or a costly task. RTF-files can be saved with almost every word processor, and importing and exporting them is easy even when using open source tools. The format of the UI specifications should be the use case document template presented in section 2.4.1. For now, it should be enough that single small applications can be imported from the specifications and that models representing single small applications can be exported to UI specifications. Later on, the possibility of the multi level UI specifications presented in Chapter 3 should be taken into consideration.

One important feature of the tool is importing of existing test cases to models, because it may be possible that there already exists even thousands of test cases generated the traditional way. This feature helps a lot when the organization is moving towards model based testing, since the old test cases can be easily used as seeds when starting to build the use case or application models. Since ASTE is used

as the test automation system, the test cases are imported from the ASTE log files that contain all the information needed.

4.4.3 Test Case Generation and Execution

There are two ways to generate test cases, either by generating random path from the graph by using probabilities (as shown in Figure 15) or by generating a path which visits every event in the graph (Figure 16). The result of path creation process is a simple list of events visited during the path. Since events contain the needed test automation information, the test script can be generated easily.

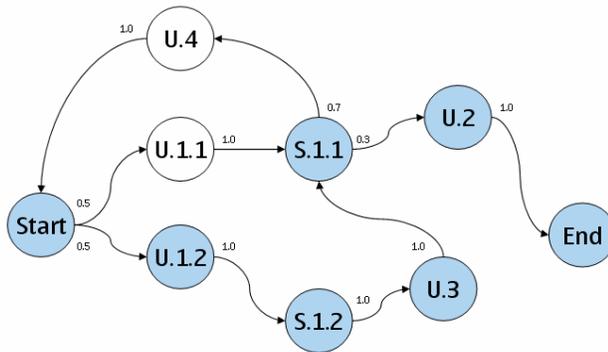


Figure 15: A random path

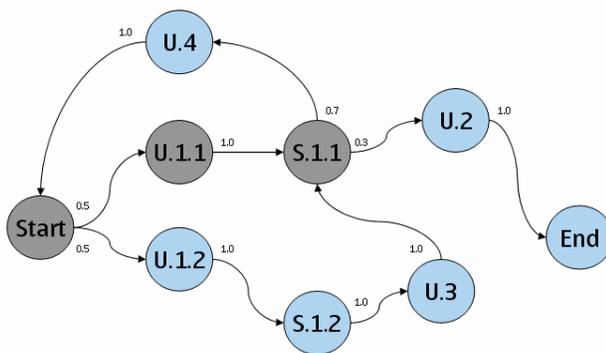


Figure 16: A path containing every event in the graph (events visited twice are marked with grey colour)

Test scripts are generated as test case files, and then sent to execution in ASTE via a socket interface. It would also be possible to send the test scripts to ASTE one keyword at a time monitoring the execution process in real time, but that possibility is not used since the tool's main target is test case generation, not execution.

4.4.4 Graphical User Interface

The most important requirement of graphical user interface (GUI) design is good usability. International Organization for Standardization (ISO) defines usability in standard 9241 part 11 as “the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.” According to usability consultant Jakob Nielsen [17], usability is composed of the following factors:

- Easy to learn
- Efficient to use
- Easy to remember
- Few errors
- Subjectively pleasing

As the ISO definition says, the users of the tool and the goals they want to achieve must be defined first in order to achieve good usability. The users of the tool can be narrowed down to two groups: UI designers and testers.

The goal of the UI designers is to be able to design the UI from a scratch, without too much hassle and extra work. The designers do not want to bother with test related details, such as test automation parameters or flags, and in the optimal situation, all irrelevant GUI elements should be hidden from the designers.

Testers' goal is to be able to easily generate and execute test cases from the already existing models. As that implies, in general testers should not have to create models themselves from a scratch, but to use the work of the designers and only add the test automation related parameters to the model. However, if the testers want to create models, it should be possible also for them in an easy to use way.

4.5 The Implementation: KENDO

The model based testing tool, KENDO, was implemented entirely with Java programming language using IntelliJ IDEA 5.1 integrated development environment. Java was chosen as the language because of the portability features it offers; compiled programs consist of so-called byte code that is run on the Java Virtual Machine (JVM), as opposed to native assembly code produced by e.g. C- or C++-compilers. Because of JVM, programs coded with Java can be run on any operating system for which the JVM is available.

Several outside libraries were needed in order not to reinvent the wheel:

- **JGraph** is an open-source Java graph component, and it is used not only as the actual graph model, but also as graph visualization in the GUI. One of the main reasons for choosing this component was that it is extremely flexible; with the component, it was possible to modify almost everything from the model level to the visualization to suit the needs of the tool.
- **Apache Velocity** is a template engine, which permits the use of a powerful template language to reference objects defined in Java code. It is used for saving the models to XML-files.
- **Apache Jakarta Digester** is used for loading the models from XML-files. It is a rule-based XML reader. Defined rules trigger defined actions when certain XML elements are encountered by the reader. For example, in the tool, new objects are created and their properties set during the XML reading.
- **iText** is mainly a library intended for exporting PDF-files from Java, but it also supports RTF-file export and that is what the library is used to perform.
- **JGoodies Looks** is used to enhance the UI of the tool.
- **ASTE** library was taken along in order to use the socket interface to send test cases for execution. The interface uses its own protocol, and the ASTE library provides simple methods for using it without the need of writing complex socket related code.

- **Apache Log4j** is used for logging events that happen during the program execution. Those events include e.g. unrecoverable error situations, warnings, and exceptions and, if enabled, debug statements.

In addition to the libraries used, the source code of KENDO consists of 70 classes, which total to approximately 11500 lines of code². The lines are divided between the different modules as follows:

- GUI uses 6000 lines of code, which contain e.g. the normal windows, file trees, tool bars and menus, as well as the graph component.
- The model consists of 3000 lines, which includes e.g. test data, flags, and the graph representing the finite-state machine. Most of the lines, approximately half, are used for test case generation algorithms.
- Utility classes, e.g. file input and output and test system interface, are made of 2300 lines of code.
- The main function, which starts the program execution, takes 250 lines of code. It contains e.g. command-line parsing, splash window and initialization functions.

The implementations of the test case generation and the graphical user interface are presented next. Requirements for them were presented in sections 4.4.3 and 4.4.4, respectively.

4.5.1 Test Case Generation Techniques

The most essential part of KENDO is the test case creation, where the model is transferred to a test script runnable by ASTE. The test script is created by traversing the graph through some path from the start event to the end. Paths can be created in many ways, but the most important of them are:

² Although all of the source code lines were written by the author, some of the classes are based on the previous version of the tool. In those cases, the interfaces were taken from the previous version but all of the functionality was rewritten from a scratch for this version.

- Random path based on the probabilities defined in the model
- Path visiting all transitions in the graph
- Path using multiple different application models

Creating the paths is not always as trivial as it sounds. Problems arise when the graph contains so-called back loops. Back loop could be best described as a transition from a lower level in a tree back to a higher level. The back loops mean that traditional graph traversal algorithms cannot be used, because of the infinite loops they cause when it is wanted that the traversal starts from a defined point and ends to a defined point, as is the case with test case generation.

If a human looks at the picture, it is easy to determine a path that visits every event from the start event to the defined end event without looping infinitely, but with computers, it is a completely different case. The graph traversal methods used in KENDO are described next.

4.5.1.1 GRAPH TRAVERSAL

Test script creation in KENDO is simple. Events in a graph are visited from the start event to the end with some graph traversal method, and the test automation keywords found in the events are written into the test script, in the order in which they are found. The test script is finalized by writing header fields required by ASTE into it.

A simple traversal method is used when a random path using probabilities is wanted. The graph is traversed from the start event to the end by selecting the transitions to follow from each event randomly. The following algorithm is used:

- 1) In the case that the model contains static flags, remove the events they disable and the transitions to/from those events.
- 2) Go to the start event.
- 3) Record the keywords in the current event and possibly set dynamic flags on/off.

- 4) Examine the events where it is possible to go from the current event. If the events are disabled by the dynamic flags, remove the transitions leading to those events from the list of possible transitions.
- 5) If only one outgoing transition in the list of possible transitions, go to the event where the transition leads and go to 3). If more than one outgoing transition in the list of transitions, choose one of them by using random number generator and the probabilities of the outgoing transitions, go to the event where the randomly chosen transition leads to, and go to 2). If no outgoing transitions, go to 6).
- 6) Currently in the last event, record the keywords and write the test script.

A more advanced algorithm is needed when 100% model coverage is wanted, i.e., that the path goes through every transition in the graph. The problem is known as the Chinese Postman Problem (CPP) [18]. A similar problem, visiting all the nodes in a graph, is known as the Travelling Salesman Problem (TSP) [19], but it is not as useful in model based testing as CPP. This is because in general it is more useful to know that every single user action possible leads to wanted system response in contrast to knowing that some, randomly chosen user actions lead to the wanted result.

Many papers published on model based testing propose different methods for graph traversal, but in general, all of them fail to present an actual working solution for the CPP. For example, in [2] there is a nice and high level collection of theories of graph traversal techniques, but it does not show in any way how the techniques presented should or could be applied to real life model based testing applications. In contrast to high-level presentations, in [20], an out-of-the-box and detailed Java implementation of an algorithm, which solves the CPP, is presented. The description of the algorithm is out of the scope for this thesis, but the presented implementation is used with some modifications in KENDO. It was thoroughly tested that the algorithm actually produces usable paths.

Since the algorithm works only with static graphs [20], and the graph model in KENDO may contain flags, which enable or disable transitions during the graph traversal, a way is needed to transform a possible dynamic graph into several static

graphs somehow. This is achieved with a simple algorithm; every possible combination of dynamic flags is created and as many new graphs there are, different combinations are created. After that, every graph created is traversed with their corresponding flag combination and events are enabled or disabled statically according to the combination. When the creation of the new graphs is done, CPP is solved for each of them one by one.

4.5.1.2 MULTIPLE MODELS

The ultimate goal of bringing model based testing into S60 software engineering process is to model the whole S60 UI. There is no sense in creating one extremely big model of the whole UI, since already the smaller applications can be complex and the state space explosion mentioned in section 2.6.2 is one of the biggest problems of model based testing.

Solution to problem is to make an abstraction of the model so that instead of displaying all of the states the application contains, only one state that represents the whole application is displayed. The new state contains at least one input and one output, which represent the start and end states of the abstracted model. Transitions can be added to and from the state normally. This is the implementation of the UI specification proposition described in Chapter 3. The abstracted state represents a small application, and the model of the whole S60 UI containing lots of different small applications represents the high level UI specification explained in Chapter 3.

In addition to the mentioned input and output, the applications may contain several other inputs and outputs because it is possible that the application can be entered and exited from different points. Good example of this behaviour is the sending of multimedia messages. When writing the message from a scratch, the MMS-application is in the normal initial state (Figure 17) where an empty text field is visible. It is also possible for the user to take a picture with the phone's camera and select "Send as MMS" directly from the camera-application. In that case, the MMS-application is entered to a state where the picture taken is already attached to the message (Figure 18).

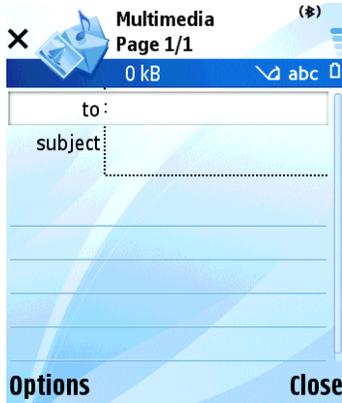


Figure 17: MMS from a scratch



Figure 18: MMS from the camera

As an example, the abstraction is visualized with Figure 19 and Figure 20. Figure 19 contains an extremely simple example MMS model, which starts normally from the creation of a new MMS in the Messaging application, but it can be entered also from the Camera application. The model can be exited either normally or back to the Camera application.

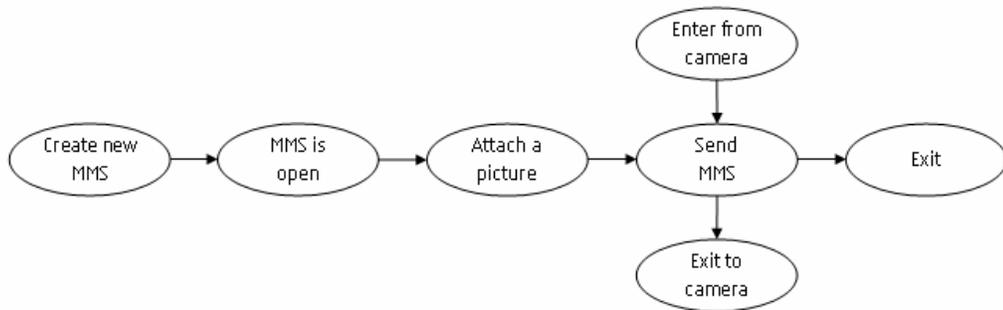


Figure 19: An example MMS model

Figure 20 contains the abstraction of the presented MMS model. It has two inputs and outputs, just like the original model. As can be seen, the abstraction hides many details, and improves the readability and usability of the model.

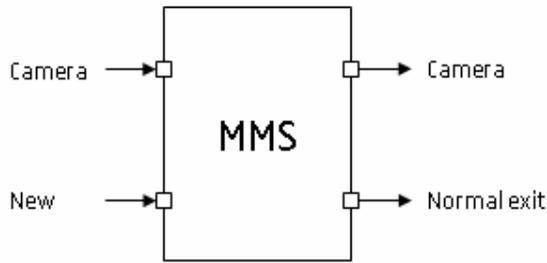


Figure 20: An example MMS abstraction

The test case generation when using multiple models is similar to the case with only one model, as the same algorithms are used in both cases. Only difference is that the wanted input and output have to be considered.

4.5.2 Graphical User Interface

The graphical user interface of KENDO is presented in Figure 21. The main GUI is divided into two tabs; the design panel and the test run panel. Design panel is the more important one of these two and only it will be discussed, whereas the test run panel only contains necessary components to execute many test cases in a row and it will not be handled at all.

The most important parts of the design panel are the graph editor, which is located in the upper-right part of Figure 21, and the event editor in the lower-right part. Graph editor allows following functions:

- Insertion and removal of events and transitions.
- Optimization, or automatic event aligning, of the graph.
- Zooming of the graph.
- Cut, copy and paste operations on events and transitions.
- Normal undo and redo operations.

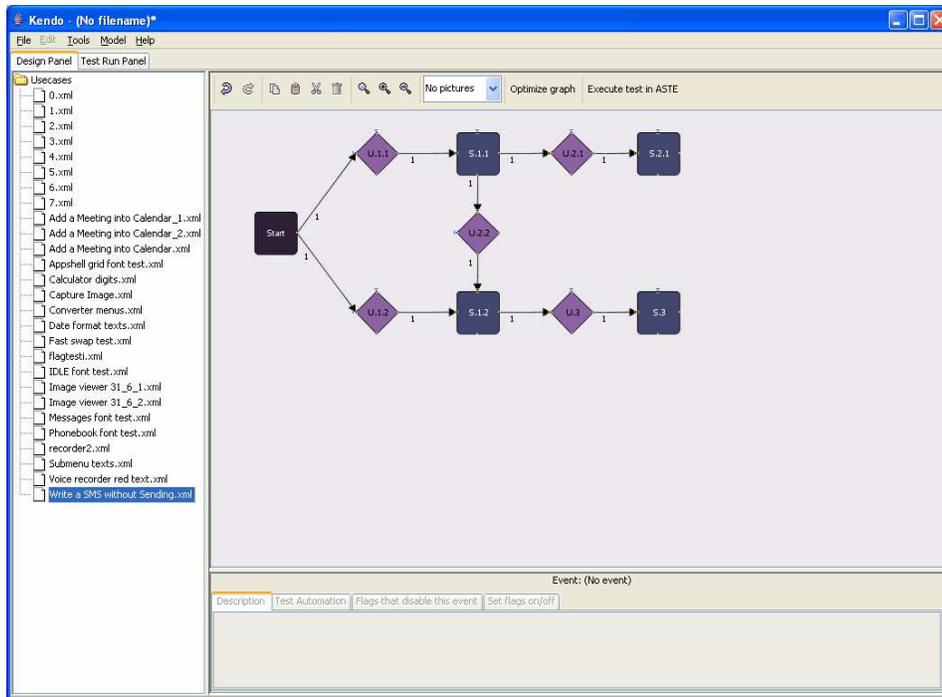


Figure 21: KENDO GUI

When user selects an event in the graph editor, its properties are opened in the event editor.

The event editor is located at the lower part of the GUI. It contains separate editors for the description, test automation, and flag parameters. Description is edited with a normal text field and test automation and flags with tables. The description, test automation, and flag editors are separated to different tabs in the event editor.

4.6 Future Improvement Suggestions

In the future, a lot of effort should be put into improving the version control of the models, which means controlling the model changes when, e.g., the UI specification changes. Currently, the import functions of the tool simply rewrite the whole model instead of only the changes, which is not desirable situation because the test automation parameters added to the model will be lost during the import. A lot of thought should be put to this feature since it is essential in the situation when it is wanted that the model be in harmony with the specification.

In addition, the problem with state space explosion described in section 2.6.2 should be studied more and this part of the tool improved. Currently features such as test data support, flags, and usage of multiple models are helpers in this matter, but a lot of room for improvements still exists.

5. Analysis

In this chapter, the model based testing tool implemented in this thesis is analyzed in order to gain some information and knowledge whether the approach taken is towards the right direction. Since this thesis is continuation to previous work, the differences between this and the preceding thesis are presented first and the impact of the changes is discussed. After that, the results achieved with model based testing are compared to the results achieved with traditional, manual testing.

5.1 Comparison to Previous Work

In [8], a use case based approach to automated user interface testing was presented. It included also a predecessor to the tool designed and implemented in this thesis. The purpose of this section is to compare the work done in this thesis to the preceding thesis.

5.1.1 Differences to the Previous Work

The biggest difference between this and the previous work on the same subject is in the overall approach taken to model based testing. In [8], use case approach was used. As explained earlier in section 2.4.1, use cases describe some functionality of a system from the user's point of view. The use cases can contain different paths, but in general, they consist of a direct flow of user and system events due to their nature, without the possibility of the flow jumping e.g. backwards or to a different path. This approach sets restrictions on the possible scenarios, since, e.g. the modelling of a whole application is not possible because of the restrictions in the flow control. In addition, use cases are for modelling of simple and precise user actions, not whole applications.

The approach taken in this thesis conforms more with the ideology of model based testing than the use case approach used before. The tool implemented can be used to model entire applications at the user interface level, which possibly allows more thorough testing of applications since there is no need for lots of separate use cases to cover the functionality of an application; a set of test cases covering the whole application can be generated from one model.

The differences in approaches are not the only factor which affects the test case generation since the tool implemented in this thesis has also several other advantages over the one implemented previously. These are explored next.

5.1.2 Advantages of the Current Tool Implementation

The following features and improvements were needed in order to enable the usage of the true model based testing ideology:

- The previous version was able to generate test cases covering all the paths in one use case and prioritize them to high, medium, or low priorities. Test case generation was based on a simple DFS algorithm, which worked well because no loops were allowed. The current version can generate one test case that goes through every transition in the model by using a modified Chinese postman algorithm, reaching 100% model coverage. It is also possible to use probabilities to create test cases randomly to simulate human behaviour. Because of the 100% model coverage it is possible to compare the code coverage achieved with model based testing to the coverage achieved with manual tests and thus gain knowledge of whether the model based approach has advantages over traditional testing methods or not. The methods are compared in the next section.
- The possibility to use test data was added, which means that the work of the testers becomes easier because they no longer need to write many separate test scripts to, e.g., test such a case as described in section 4.4.1.2. The test data also reduces the number of states required in an application model.

- In order to reduce the state space explosion, which is one of the biggest problems of model based testing as described in section 2.6.2, flags were added to the model. Flags can be used to, e.g., customize the same model for different devices (the camera example in section 4.4.1.1) which means that fewer states are needed to create the model.
- The usage of multiple models when creating the test cases was added to the tool. This allows the possibility of creating one big model representing e.g. the whole S60 UI from smaller application models. It also allows the possibility of model import from and export to UI specifications conforming to the proposition presented in Chapter 3.

The changes done to this version allow the usage of the full potential of model based testing, in contrast to the previous version, which was basically a helper for creating use case based test cases. In addition, some of the features added aim to reduce the problem of state space explosion, which can prove to be a valuable feature in the future.

In [8] it was shown that a graphical method of creating test cases is a lot faster than a manual one. The next section aims to prove that model based testing is a technique that brings financial benefits to an organization if its whole power is unleashed.

5.2 Comparison Between Testing Methods

In this section model based testing is compared to traditional, manual testing in terms of the code coverage reached. At first, the used test environment is explained, and after that, the test methods and the results they produced are analyzed.

5.2.1 Used Test Metrics

There are many different ways to measure code coverage, and the way used during these tests was the so-called decision coverage, which measures conditional expression evaluations (true or false), case branches in switch statements, exception catching and control transfers. For example, let us consider the C code below:

```
int positive(int integer) {
    if(integer >= 0)
        return 1;
    else
        return 0;
}
```

The decision coverage criterion is only met if the function is called with both negative and positive integers because of the if-else block.

5.2.2 Test Environment

Comparison between traditional, manual testing and model based testing was done with the S60 applications Image Viewer and Voice Recorder by the Nokia S60 multimedia testing group. The Symbian code was instrumented using CTC++ code coverage analyzer by Testwell (<http://www.testwell.fi>). CTC++ uses source code instrumentation, which means that special instrumentation statements are added to the source code during pre-processing. After that, the source code is compiled and the CTC++ run-time library is linked to the instrumented program. During the actual program execution, CTC++ collects the code coverage history in memory and writes it to a data file after the execution is over. The data file can then be used to analyze the actual code coverage parameters.

Manual tests are executed by testers based on test cases created from use case descriptions. The use case format used differs a little bit from the one described in section 2.4.1. The use cases contain test objectives (“Test that an image opens correctly”), required test data (“One image of any supported image format”), preconditions (“There is one image available on the phone”) and the execution steps which contain the description of what to do (“Open the image”) and what the expected result is (“The image is opened correctly”). The tester’s job is to read the test cases, push the buttons on the actual device, and report the results achieved.

The models used in this comparison were created by hand by a tester because no suitable user interface design documents were available in the organization at the time. Although this method is against everything said about model creation throughout this thesis, it is acceptable within the scope of this comparison. The test cases were generated from the models randomly, and executed automatically for 8

hours. The automatic and manual executions can be compared, since the test scripts used when automating the tests are based on the manual test cases.

Next, the applications used in this comparison are presented.

5.2.2.1 IMAGE VIEWER

Image Viewer application is used to view images on the phone. Images can be captured with a camera, received by multimedia messages, or transferred by other means to the phone.

The application contains quite a few features, the most important being the ability to send images via multimedia messages or through Bluetooth or infrared connections. When sending images via messages, the image is possibly resized or repacked in order to reduce the size of the image file due to multimedia message restrictions. Other connections do not require altering the image in any way; it is sent as is.

Other features of the application include setting an image as the background image of the phone and rotating, zooming, renaming and deleting an image. Images can also be browsed with the application.

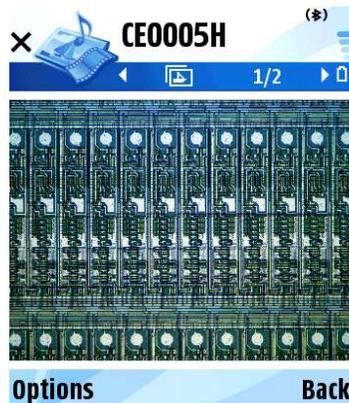


Figure 22: Image Viewer

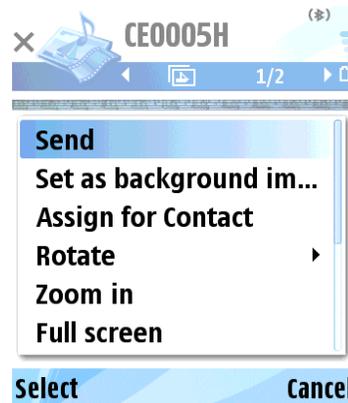


Figure 23: Image Viewer options

The Symbian source code of the application is divided into 16 source files that contain 287 functions and 7741 lines of code. The number of lines is high in light of

the apparent simplicity of the Image Viewer. CTC++ identified 1840 measurement points for code coverage measurement.

5.2.2.2 VOICE RECORDER

Voice Recorder is used to record short sound clips from the phone's microphone. The application can only be used to record new sound clips; it is not possible to record more sound to previously recorded sound clips. Another restriction applies to playback; since the application is a recorder, only the newly recorded clip can be played back. Other playback control functions include pausing and stopping, fast forward and fast reverse. The volume of the playback can also be set. The recording quality can be either high or low. The latter is for sending the recorded clip via a multimedia message, because of the restrictions in the size of multimedia messages.

After the recording, the clip is automatically saved either to the phone's internal memory or to a memory card. It is also possible to send the clip immediately via a multimedia message. Clips cannot be deleted from the recorder directly; for that purpose, some form of a file browser must be used. In addition, the playback of old recordings is done via a browser.

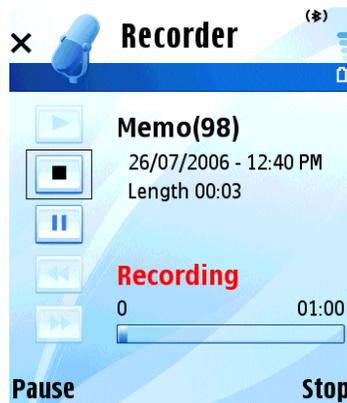


Figure 24: Voice Recorder recording



Figure 25: Voice Recorder when recording finished

Voice Recorder is a simple application, but still the source code base is large; it is divided into 32 source files that together contain 364 functions and 9019 lines of

code, which is more than the Image Viewer application has. 2234 measurement points were used by CTC++.

5.2.3 Results

The results of the comparison are presented in Table 2. The table contains the execution time used and the code coverage percentage reached with both model based testing and the traditional method. The measured code coverage percentage, which tells us how much of the application source code is executed during testing, is used as the comparison figure between the two methods.

Table 2: Results of the comparison

Application	Figures	Model based testing	Manual testing
Image Viewer	Execution time	8	48
	Code coverage	71%	79%
Voice Recorder	Execution time	8	15
	Code coverage	55%	60%

The table shows that the code coverage achieved with model based testing is not far from the coverage achieved with the traditional method. This is an interesting and a promising result, especially when considering that the models were built by hand, in contrast to being imported from any design documents. This means that the models might contain small errors or that they are not as complete as they could be. In addition to the manual creation of models, two other big factors that affect the code coverage reached are:

- The models do not contain any inputs from or outputs to different applications, in contrast to use cases that might contain, e.g., a case where the image in Image Viewer or the sound clip recorded in Voice Recorder is sent via MMS. In these cases, the corresponding application code is not executed when using model based testing.

- The use cases contain cases that are not pure user interface tests; e.g., they may test performance and memory consumption. This means that more of the application code is executed with the tests based on the use cases since the model is only a representation of the user interface functionality.

What is even more interesting than the code coverage achieved by model based testing is that a huge number of man hours were used for the execution of traditional tests. The model was executed with KENDO completely automatically for 8 hours with random traversal of the graph.

The two testing methods can be best compared with code coverage / execution time ratio, which tells us how many percent the code coverage percentage increases for every hour used for test execution. For the Image Viewer the ratio is 8.9 for model based testing and 1.6 for the traditional method, which shows a clear difference and advantage for the model based approach. The ratios of Voice Recorder show the same trend, as they are 6.9 for model based testing and 4.0 for the traditional method.

Because the test execution with KENDO was completely automatic, the execution did not cost anything in contrast to the man hours used for the traditional test execution. It is also considerably faster; in the case of the Image Viewer, almost the same code coverage was reached with 8 hours of automatic testing as with 48 hours of manual test execution. This means that over one week of work can be reduced to a simple, overnight, automated task.

When comparing the time used to create the test cases and the models the advantages of model based testing become even clearer, if possible. Both of the models were created in approximately 8 hours, and debugged for another 32 hours, which makes the total creation time of approximately 40 hours. No exact figures are available for the manual test case creation, but it was approximated that the absolute minimum is one hour per test case, and since about 150 test cases are used per application, the total manual test creation time is approximately 150 hours. Model based testing is a clear winner also in terms of the man-hours used to start-up the testing.

Because the time spent on both test case creation and test execution is known on both testing methods, it is easy to compare them also on yearly basis. About 30% of the

manual test cases are run with each software release, which are bi-weekly. This means that in the case of the Image Viewer, about 50 hours of testing will be conducted on 26 weeks of a year, which totals to 1300 hours per year of test execution. Altogether, about 1450 hours is spent on manual testing per year when summing the execution time with the time spent on test case creation. The same figure for the Voice Recorder is 280 hours per year.

By using model based testing, the time spent on testing activities would be significantly lower. With the same testing cycle, i.e., bi-weekly, the testing would take about 248 hours per year, including the model creation, and would be more thorough because of the fact that only a subset of manual test cases are executed during the bi-weekly cycle. The full set, which reached the higher code coverage in the comparison, is executed rarely. It also has to be remembered that the test execution is fully automated in the case of model based testing. The yearly figure for model based testing does not contain the maintenance of the model; it cannot even be estimated, since far too little experience on the subject has been gained. If the goal presented in section 3.1, automatic model import from user interface specifications is reached, then the man-hours required for the model creation and maintenance would be reduced. This means that model based testing would be even clearer winner, and it is a shame that this could not be proven empirically due to the lack of formal enough user interface specifications.

The above calculations show that by using model based testing, approximately over one thousand man-hours would be saved per year in the case of the Image Viewer. The Voice Recorder did not show as much man-hour savings, but still fully automated testing is significantly cheaper than 130 hours of manual testing. These figures show clearly, that *model based testing reduces the time, labour and money needed to perform the testing activities.*

6. Conclusions

In this chapter, the results obtained from this work are summed up by reviewing and assessing them, and explaining how they are exploited currently in Nokia organization. The chapter is concluded with a discussion about the future research possibilities.

6.1 Results

The comparison made between model based testing and traditional, manual testing showed clearly that the former is the cheaper and faster method, and almost as effective in terms of the code coverage reached as the latter one. Model based testing was a little bit behind the traditional method in code coverage, but the possible reasons for this were explained in section 5.2.3.

It was approximated that on yearly basis, model based testing would save significant amount of man-hours per S60 application, and in the case of the Image Viewer, over one thousand.

6.2 Assessment of Results

The results obtained in the comparison between testing methods should be similar in other cases as well, as long as the system under test is a rather simple user interface. The code coverage reached of course depends on the quality of the model, and nothing can be said about the manual test case coverage on other systems since it depends heavily on the quality and the thoroughness of the test cases. In the case of other S60 applications than Image Viewer and Voice Recorder, the results in terms of

the code coverage should follow the results achieved in this work because of similar qualities between the manual test cases.

The man-hour savings calculated in section 5.2.3 are approximations, but the figures used as the starting values were more under- than overestimations. Similar approximated results should be achievable with other cases as well.

6.3 Exploitation of Results

Currently the tool implemented in this work is in continuous use in Nokia S60 multimedia testing group, and the usage might also be extended to other groups as well, but the future is not clear at this stage. The promising results obtained in the comparison of testing methods should be a clear signal for the organization that the model based approach should be exploited even more, and that time and money should be invested into it.

Currently it seems not possible to deploy the model based approach to the organization with the starting point suggested in Chapter 3, which uses the user interface specifications as the basis for the model creation. This is because the change would involve too many people, and in an organization as big as Nokia, it seems to be an impossible task to change the ways of working across the department borders. This is mainly because a part of software engineering process as big as the user interface specifications will not be modified unless a good reason exists, and no good reason has been introduced yet. Hopefully the results obtained in this work could be one.

Another reason for the difficulties in deploying the model based approach is the lack of productization of the KENDO tool implemented in this work. It is hard to sell the approach to different departments when the tool, which is supposed to be used, does not contain any documentation and on the top of that is still quite hard to use.

The difficulties are a shame, because in this work it was proven that model based approach brings benefits into the user interface level testing, and the only thing left missing was formal enough user interface specifications from which the models could have been imported. If those were available, the comparison between testing methods would have been more precise because the models would have been accurate

representations of the user interface, in contrast to the hand-crafted models used in the comparison.

6.4 Future Research

The scope of this thesis was on the user interface level, and as said in section 3.2, models should be brought to other levels of software engineering as well. This has some problems, as the user interface approach used is a special case and cannot be generalized to other levels easily because of the simplicity of the finite state machines used. The other levels require more complex model representations.

At least the linking of the architecture design and the integration testing with a model should be researched thoroughly, and added to the tool implementation. The module design and module testing phases can also be studied, but currently it seems that adding those to the same tool is not a good idea, because it will probably bloat the models into a non-usable level.

Model based testing is an excellent tool for certain tasks, e.g. for small applications, embedded systems, user interfaces and state-rich systems as stated in section 2.6.1, but it cannot be effectively used for everything. For example, module design and testing seem to be too complex to be integrated into the model based approach.

References

- [1] Fewster, M. & Graham, D. *Software Test Automation*. Harlow, United Kingdom: Pearson Education Ltd, 1999. 574 p. ISBN 0-201-33140-3.
- [2] Robinson, H. *Graph Theory Techniques in Model-Based Testing*. 16th International Conference and Exposition on Testing Computer Software, Los Angeles, California, USA, 1999.
- [3] Beizer, B. *Software Testing Techniques*. 2nd Ed. New York, USA: Van Nostrand Reinhold Co, 1990. 550 p. ISBN 0-442-20672-0.
- [4] El-Far, I. K. & Whittaker, J. A. *Model-based Software Testing*. In: Marciniak, J. (ed.), *Encyclopedia on Software Engineering*, Volume 1. New York, USA: John Wiley & Sons Inc, 2001. pp. 825-837. ISBN 0-471-21008-0.
- [5] Robinson, H. *Finite state model-based testing on a shoestring*. International Conference on Software Testing Analysis and Review, San Jose, California, USA, 1999.
- [6] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R. & Stauner, T. *One Evaluation of Model-Based Testing and its Automation*. In: *Proceedings of the 27th International Conference on Software Engineering*. St. Louis, MO, USA. 15.-21.5.2005. New York, NY, USA: The Association for Computing Machinery. pp. 392-402. ISBN 1-59593-963-2.
- [7] Apfelbaum, L. & Doyle, J. *Model Based Testing*. 10th International Software Quality Week, San Francisco, California, USA, 1997.

- [8] Nikkanen, M. *Use Case Based Automatic User Interface Testing in Mobile Devices*. Master's thesis. Helsinki University of Technology, Department of Computer Science and Engineering. Espoo. 2005. 79 p.
- [9] Collard, R. *Test Design: Developing test cases from use cases*. Software Testing & Quality Engineering Magazine, 1999. Vol. 1:4. pp. 30-37.
- [10] Prowell, S. J. *TML: a description language for Markov chain usage models*. Information and Software Technology, 2000. Vol. 42:12. p. 825-833.
- [11] Selic, B. *The Pragmatics of Model-Driven Development*. IEEE Software, 2003. Vol. 20:5. pp. 19-25.
- [12] Mellor, S. J., Clark, A. N. & Futagami, T. *Model-Driven Development*. IEEE Software, 2003. Vol. 20:5. pp. 14-18.
- [13] Blackburn, M., Busser, R. & Nauman, A. *Why Model-Based Test Automation is Different and What You Should Know to Get Started*. International Conference on Practical Software Quality and Testing, Washington, USA, 2004.
- [14] Safford, E. *Test Automation Framework, State-based and Signal Flow Examples*. Twelfth Annual Software Technology Conference, Salt Lake City, USA, 2000.
- [15] Utting, M. & Legeard, B. *Practical Model-Based Testing: A Tools Approach*. 1st ed. San Francisco, CA, USA: Morgan Kauffmann, 2007. 456 p. ISBN 0-12-372501-1.
- [16] Nicol, G. T. (Ed.). *XTND – XML Transition Network Definition*. W3C Note, 21.11.2000. [Cited 22.3.2007]. Available at: <http://www.w3.org/TR/xtnd/>.
- [17] Nielsen, J. *Usability Engineering*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 1993. 362 p. ISBN 0-12-518406-9.

- [18] Eiselt, H.A., Gendreau, M. & Laporte, G. *Arc Routing Problems, Part I: The Chinese Postman Problem*. Operations Research, 1995. Vol 43:2. pp. 231-242.
- [19] Kruskal, J. B., Jr. *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*. Proceedings of the American Mathematical Society, 1956. Vol 7:1. pp. 48-50.
- [20] Thimbleby, H. *The directed Chinese Postman Problem*. Software – Practice and Experience, 2003. Vol 33:11. pp. 1081-1096.