# Model-based testing

Joonas Lindholm

Helsinki November 1, 2006

UNIVERSITY OF HELSINKI
Department of Computer Science

# Contents

# 1 Introduction

Software testing is the task of executing software following a specifig sequence of steps, comparing the software's actual and expected outputs. In order to expose as many bugs as possible with the tests, the tester must first thoroughly understand what the software is supposed to do. This can be a difficult task due to the very nature of software, as witnessed by its intangibility and complexity compared to products of traditional engineering. Written specifications and requirements, are often incomplete and ambiguous, if they even exist. So when a software tester selects which tests to run and what kind of problems to look out for when executing the tests, he usually depends on his own mental picture and understanding, a *model*, of the software. As Kaner et al [KBP01] state, every kind of testing could therefore be called *model-based*. When comparing different testing styles, only the explicitness and lifetime of the model varies, but a model still exists.

In the last two decades, modeling has become a popular technique in designing and developing software, practiced in one form or another almost anywhere software is being made. Object oriented programming practices and the establishment of UML as a standard for object oriented design, have contributed to the increasing reliance on modeling in the field.

The abundance of modeling techniques in software design as well as the fact that modeling already is an implicit a part of software testing, has lead to the definition of a set of testing practices, commonly labeled as *model-based testing*. El-Far and Whittaker [EFW01] describe model-based testing, or MBT, as a style of software testing which encourages the use of explicitly written down, sharable and reusable models for directing and even automating the selection and execution of test cases and evaluating the results.

Model-based testing can originally be traced back to hardware testing, mainly in the telecommunications and avionics industries [RR00]. It is nowadays well established in the academic field of software testing but is also slowly gaining interest in the software industry. In this paper model-based testing is described from a *black-box testing* perspective applied in the *system testing* of software, which is the way it's treated by most of the academic research.

A brief introduction on modeling is presented in section 2, continuing with an outline of the model-based test activities in 3. The last two chapters describe how a testable model is created in practice, and how a test suite can be derived from it.

# 2 Modeling software behavior

Modeling, whether in software engineering or any other discipline, means the process of creating a representation of a real system or phenomenon, abstracting and simplifying its behavior. Models always describe the system or phenomenon from a certain viewpoint. Two totally different models may still describe the same system, depending on which aspect of the system it is used to depict. In case of a software system, different kind of models can be used to describe for instance the control flows in the source code, module dependencies or GUI states. The models may focus on anything between internal logic to the user perceivable input and output behaviour. Models can be presented using any of numerous different notations, from graphical charts (such as UML) to mathematical models or specification languages (such as SDL and Z). As El-Far and Whittaker [EFW01] point out, a model should be "as formal as it is practical", making it important that a right kind of modeling technique is selected for the application in question.

## 2.1 Finite state machines

In case of software testing, and software engineering in general, the fundamental model on which many other models are based on, is the *finite state machine.* This is obvious from the way most software systems are usually perceived, used and tested – in any moment, a system appears to always be in a specific state, in which a defined, finite set of inputs are allowed. When the system receives one of the available inputs, it performs some action and proceeds to another state, in which another set of inputs are allowed. Systems that match this description are known as *reactive systems* and for instance graphical user interfaces often fit within this category. Finite state machines are appropriate models such systems.

In automata theory [HMU79] finite state machines are formally defined as a quintuple $(Q, \Sigma, \delta, q_0, F)$, where in the case of a software system

- $Q$ is the set of software states
- $\Sigma$ is the set of possible inputs in any state
- $\delta$ is the transition function, defining what state transition occurs for each software input in any state
- $q_0$ is the software's initial start state, one of the states in $Q$

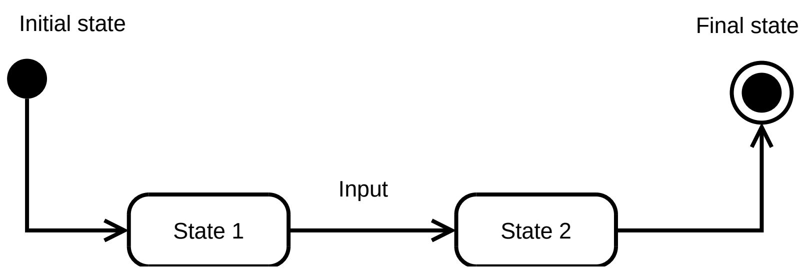

Figure 1: A finite state machine using UML notations

- $F$ is the set of final states of the software, a subset of $Q$

Finite state machines are usually represented as *transition matrices* or usually more informatively as graphical *transition diagrams*, using notations such as in figure 1.

## 2.2 Other models

The standard FSM definition can be restricted or extended so it is a better suits the needs or the environment of the modeled system. The finite state machine and its derivatives are commonly known as *transition based models* [Har06]. A classic extension of the FSM is the *statechart* which introduces hierarchy to models. Using statecharts a higher-level model can be expanded into several lower-level models, where one state in a higher-level model is further described by a separate lower-level FSM. Hierarchical statecharts make it easier to understand complex behavior. When extending the FSM model with *conditional* transitions makes it possible to model a system which behavior is not purely state based, but also dependent on context and external variables. Statecharts are included in the UML framework, where they are called *state machine diagrams.*

Other UML models can also be applied in describing a software system behavior when performing model-based testing, for instance activity diagrams and sequence diagrams. Since modeling in UML is often already performed in many software projects, reusing and adapting these models for testing purposes is encouraged. A methodology for systematically transforming software design models in UML to test specifications has been presented by Zhen [Zhe04]. Even without attempting such a formal derivation of a testable model from existing UML design models, they are still useful insights into the system's behavior.

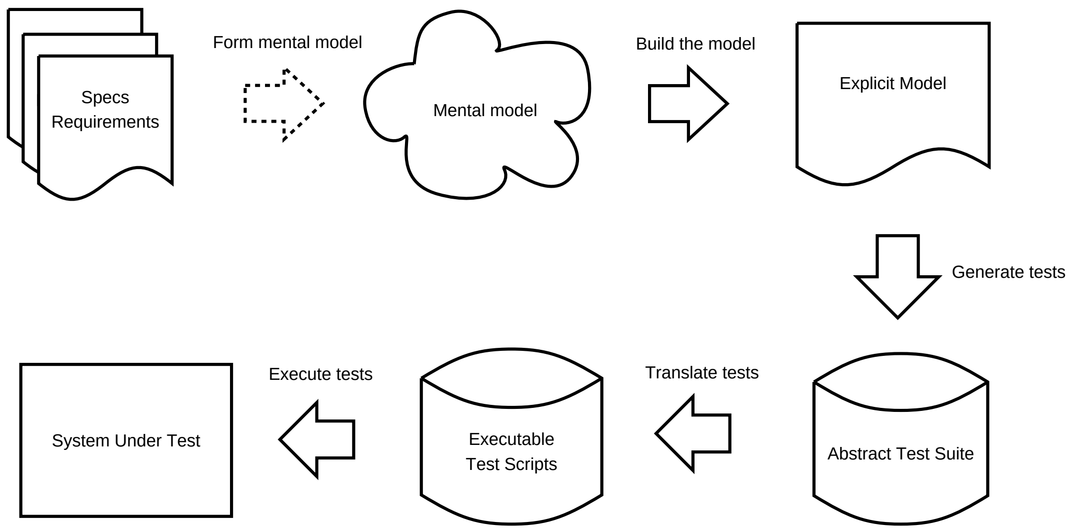

Figure 2: The main activities in the model-based test process

## 3 Model-based test activities

Engaging in model-based testing shifts the focus in the test process from the traditionally difficult and time consuming tasks of test case generation and execution, to constructing testable and accurate enough models of the software under test. Figure 2 displays the main activities in a model-based test process. The activities begin with the construction of the model, by transforming the mental, implicit models into an explicit one. In model-based testing, the test cases generated from the model are actually nothing more than different paths in the model. Creating the *abstract* test suite is an easily automatable process of simply traversing through the states and transitions in the model, until the wanted *model coverage* is met. The generated abstract tests are then translated to an executable test scripts, which can run on the system under test. By comparing the actual and expected outputs of the system against the model, it can be decided whether the test passed or failed.

After one iteration of the process has been performed, the results are analyzed. The possible actions include generating more test cases using the old model, modifying the model allowing new kind of tests to be generated, or finally stopping the testing if it's estimated that sufficient software quality has been reached.

Creating a new software system often includes rapid, unanticipated changes in the system's requirements and specifications as the development proceeds. As the software's change and evolves, it's a well known fact that manually specifying test cases and keeping the tests up-to-date with respect to a decided coverage criteria, is a difficult and time consuming task. For instance changing the behavior of a single control in the software's user interface makes every single test case using that control outdated. The tester must manually search the influenced test cases and update them. The more intricate the changes are, the more difficult the process of maintaining the test suite becomes.

Maintaining a model-based test suite, however, proves to be much simpler. When the specifications change, only the relevant parts of the models must be changed. Since the model itself prescribes the available set of test cases, and test case generation and execution can be automated, the test suite is always up-to-date and maintenance costs approach zero [Har06]. The amount of possible test cases that can be automatically generated is practically infinite, leading to execution of test cases that the testers themselves couldn't have imagined.

Engaging in model-based testing allows the testers to engage in their work in an earlier stage of the development process. The models can be developed incrementally, starting from higher-level models describing the greater software context, proceeding towards lower-level models describing actual user interface details, or even tracing the execution at source code level. Rosaria and Robinson [RR00] also point out that commitment to model-based testing can help find bugs already before entering the system testing phase. Simply doing the modeling, before creating or running any tests, should lead to improved software quality. Modeling helps to find inconsistencies in the specification and design before implementation, which usually makes it easier to fix the problems.

# 4 Creating a testable model

The process of developing a testable model of software roughly in three steps: *understanding the system*, *identifying the system inputs* and *building the model.*

## 4.1 Understanding the system

The process begins with the same task as with any testing style, which is acquiring of information and understanding about the system's intended functionality. There are different sources for getting the information needed, such as system requirements and specifications and existing design models. Specifications are rarely informative enough to be the only source of information in building sufficient mental understanding of the systems. By actually using and exploring the system, for instance using *exploratory testing* techniques [KBP01], and combining the perceived behavior with the written specifications, a mental model of the system begins to take form.

It is necessary to understand the software's functionality from different perspectives, analyzing its environment, different users and interfaces. The bigger and more complex the system is, the harder the task of building the model becomes. It's not possible to completely describe a whole system in a single model, so the test objectives should be narrowed down to keep the process manageable [EFW01]. Also, there's no single kind of software model that fits every purpose, so a suitable model, or collection of models, must be selected separately in every case.

## 4.2 Identifying the inputs and constructing the model

After a basic understanding of the software's functionality has been established, the modeling process continues with *enumerating the system's inputs* [RR00]. This includes identifying every control in the user interface or other interfaces, depending on the system. The allowed value domains, including boundaries and illegal values, should be analyzed for each input. The system's expected responses to the inputs are analyzed and documented similarly. In case the model-based test cases are meant to be automated, which they usually are, this is the perfect time for the testers to start investigating how these inputs can be programmatically generated and results captured through a test automation interface.

When enumerating the software's inputs, the tester should observe in which situations the input's can be applied, and how the expected responses change depending on the situation. The *model* begins to take shape when different *operational modes*, i.e. *states*, can be recognized. When the system is in a certain state, a certain set of inputs are available. When inputs are applied the system makes a transition to a different state where different inputs are available. The states, inputs and rules controlling how different inputs change the systems state are finally presented through a model. Figure 3 presents a very simple model of the standard Windows Calculator application, with a viewpoint focusing only on switching between standard and scientific modes.

By iterating through the different steps in creating the model, changing and refining the viewpoint, it is possible to learn more about the systems purpose and func-

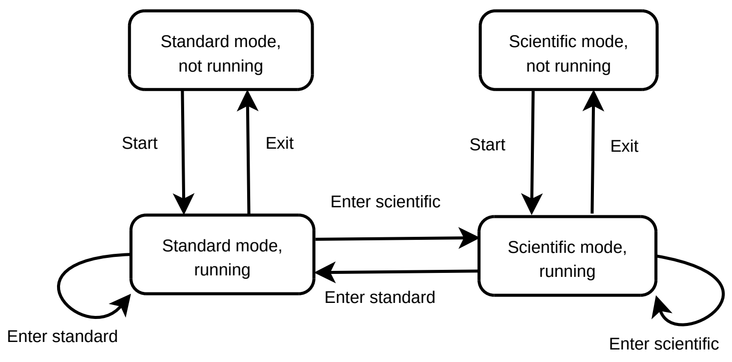

Figure 3: Windows Calculator states and transitions

tionality, and encounter more inputs and states. The model can be hierarchically divided in separate models, describing different levels of functionality, or combined in the same model for instance using *composite states* found for instance in UML state diagrams. Figure 4 extends the Windows Calculator application, by further extending the functionality within the scientific mode state.

## 4.3 State space explosion

The example depicted in the figures above are of course simple, academic "toy examples". In real model-based testing examples, the amount of states and transitions would quickly become hugely greater. This, in fact leads to the main problem facing model-based testing, namely *state space explosion.* Adding new states to already complex models can lead to quick, possibly even exponential growth of the number of states and transitions in the system. Even though test generation using the model is automated, there's a limit to how large models and resulting test suites it's practical to develop and maintain. The tester developing the model must select between a limited model, or limited test coverage in the actual system.

The state space explosion can be kept in control through two different tactics [EFW01], namely *abstraction* and *exclusion.* By opting for abstraction multiple states or inputs are combined to a single state or input. This can be observed in figure 3 where the different states within the scientific mode are all abstracted within a single state. Comparing figures 3 and 4 shows how adding a simple new layer of functionality to the model quickly increased the total size of the model.

In the case of exclusion, some information is simply left out the model. This may include excluding a large subset the total functionality from one model, and instead

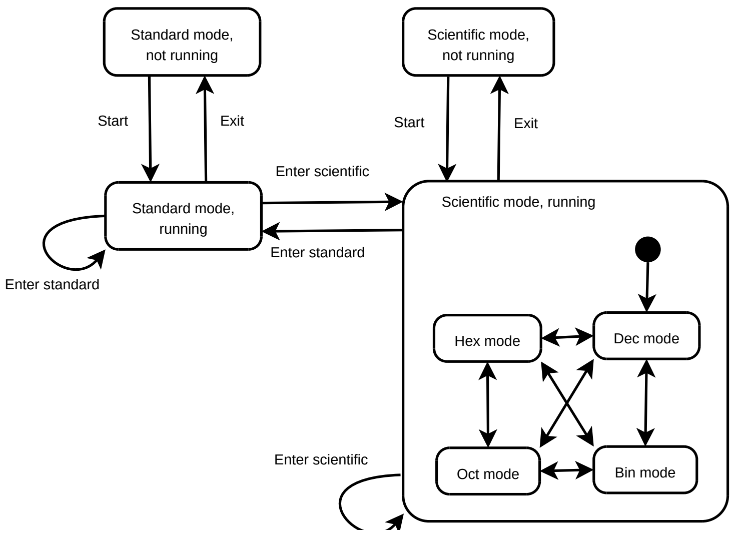

Figure 4: Windows Calculator, with scientific mode state explained using a lower level model (transition labels omitted for clarity)

placing it in an own, separate model. It's evident, that a lot of functionality has been excluded in both examples presented above, which of course should be included in some other tests.

## 5 Creating and executing the tests

After the model has been created, it's structural validity should be checked before generating the *abstract test suite*. Binder [Bin99] presents a checklist which can be followed when validating the model. Things that should be checked include verifying that the model has a unique initial state, no duplicate states and that all states are reachable in the model.

The test cases generated from models are in the form of sequences of inputs, and expected resulting states. When the model is done as finite state machines, a test case can be any valid path in the model. What kind of paths to choose in the test case depends on the wanted *coverage* in the model. An abstract test generated from the Windows Calculator model in figure 3 is displayed in table 1.

| # | **Action** | **Expected state** |
|---|---|---|
| 1 | Start | Standard mode, running |
| 2 | Enter standard | Standard mode, running |
| 3 | Enter scientific | Scientific mode, running |
| 4 | Enter scientific | Scientific mode, running |
| 5 | Exit | Scientific mode, not running |
| 6 | Start | Scientific mode, running |
| 7 | Enter standard | Standard mode, running |
| 8 | Exit | Standard mode, not running |

Table 1: Test case with full state and transition coverage of the model in figure 3

## 5.1 Test coverage

Binder [Bin99] lists possible levels of coverage that can be aimed for when designing a test suite with the help of a model:

- *All states* coverage is achieved when the test reaches every state in the model at least once. This is usually not a sufficient level of coverage, because behavior faults are only accidentally found. If there is a bug in a transition between a specific state pair, it can be missed even if all states coverage is reached.
- *All transitions* coverage is achieved when the test executes every transition in the model at least once. This automatically entails also all states coverage. Reaching all transitions coverage doesn't require that any specific sequence is executed, as long as all transitions are executed once. A bug that is revealed only when a specific sequence of transitions is executed, is missed even in this coverage level. The coverage can be increased by requiring *All $n$-transition* coverage, meaning that all possible transition sequences of $n$ or more transitions are included in the test suite.

Because a finite state machine is basically just a directed graph, the test cases can be generated using well-known graph traversal algorithms [RR00]. Examples of algorithms used are the *Chinese Postman* and *Shortest Path First* algorithms. The paths can be restricted by forcing the test to start and end in the same starting state, limiting the number of loops in the path, or the number of states that the path must visit, depending on the test needs.

## 5.2 Test execution

The generated abstract test suite is simply a set of different paths in the model. If the test are perfomed manually by a person, the generated tests are sufficient guidelines for how the test should be performed and what the expected results are. The generated test suite is usually too large for manual execution, and a key point in model-based testing is the frequent regeneration and re-running of the test suite whenever the underlying model is changed. Consequently, achieving the full potential of model-based testing requires automated test execution.

Test automation is too wide a subject to be even superficially covered here. Model-based testing requires that there is *some* programmable interface for executing the software under test, such as an API or an GUI test automation interface. The abstract test suite is translated into an executable test script using the software's available testing interface. The translation of abstract tests to executable tests can be performed after the original test suite is generated, or alternatively the translation can be embedded already into the procedure where the original abstract test suite is generated. Finally, the executable test suite is run through a *test harness* or *driver* and the actual results are compared against the model [RR00, EFW01].

An alternative way for creating and executing model-based tests is *on-the-fly testing*, described by Veanes et al [VCSK05]. On-the-fly testing integrates the derivation of test steps from a model and executing the test into a single process. The algorithm for traversing the model is combined with translating the abstract test steps into executable test inputs and the oracle evaluating the results. Every time a previously untried transition is found in the model, it can immediately be executed in the system. This is particularly useful when a bugs are found during testing. The test driver can be programmed to continue, even after detecting a bug, by excluding the faulty state or transition from the model, and attempting to reach coverage in the rest of the model.

# 6 Conclusion

Model-based testing, in all its simplicity, appears to be a useful and efficient testing method for quickly reaching large test coverage in a system, without enormous testing costs. Because model-based testing implies radical changes in the software design and testing processes, a full scale adoption of the technique requires thorough evaluation of its benefits and drawbacks, before seriously attempting to take it into use. Even though radical benefits and savings and even ten-fold productivity improvements have been reported in the academics [AD97, RR00], the reality may be different. In Hartman's [Har06] experience of model-based testing in the industry, many attempts at model-based testing resulted in outright failures, or were abandonded and never used again in the future, even after partial success in previous projects.

As long as modeling is performed one way or another in software projects, model-based testing will continue to have a certain appeal, simply because of the reuse value provided by existing design models. Another appeal of model-based testing is its flexibility, allowing a vast number of tests, with different objectives and levels of coverage, to be developed with low cost and effort, once the model is ready. An increasing number of software vendors have recently released their own, commercial model-based testing tools to the market, indicating that there might yet be a bright future for model-based testing.

## References

AD97 Apfelbaum, L. and Doyle, J., Model based testing. *Proceedings of the 10th International Software Quality Week*, May 1997, URL `http://www.geocities.com/model_based_testing/sqw97.pdf`.

Bin99 Binder, R. V., *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

EFW01 El-Far, I. K. and Whittaker, J. A., Model-based software testing. *Encyclopedia of Software Engineering*, Marciniak, J. J., editor. John Wiley & Sons, Inc., 2001, URL `http://www.geocities.com/model_based_testing/ModelBasedSoftwareTesting.pdf`.

Har06 Hartman, A., Adaptation of model based testing to industry (presentation slides). *Agile and Automated Testing Seminar*, Tampere University of Technology, Tampere, Finland, August 2006, URL `http://www.cs.tut.fi/tapahtumat/testaus06/alan.pdf`.

HMU79 Hopcroft, J. E., Motwani, R. and Ullman, J. D., *Introduction to au-*

*tomata theory, languages and computation.* Addison-Wesley, Reading, MA, USA, 1979.

KBP01 Kaner, C., Bach, J. and Pettichord, B., *Lessons Learned in Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

RR00 Rosaria, S. and Robinson, H., Applying models in your testing process. *Information and Software Technology*, 42,12(2000), pages 815–824. URL `http://www.geocities.com/harry_robinson_testing/ApplyingModels.pdf`.

VCSK05 Veanes, M., Campbell, C., Schulte, W. and Kohli, P., On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-05, Microsoft Research, January 2005. URL `http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=852`.

Zhe04 Zhen R. D., Model-driven testing with UML 2.0. *Proceedings of Second European Workshop on Model Driven Architecture*, Akehurst, D. H., editor, University of Kent at Canterbury, United Kingdom, August 2004, URL `http://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Dai.pdf`.