Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Test generation based on state machine models

MASTER'S THESIS

Author
Unicsovics Milán György

Advisor
Dr. Micskei Zoltán

November 29, 2015

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Unicsovics Milán György*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2015. november 29.

Unicsovics Milán György
hallgató

Kivonat

...

Abstract

...

Introduction

Problem and thesis statement

Software testing is an important part of any software development process, because it is one of the most popular verification technique. The main goal of software testing is fault detection, where we compare the software's intended and actual behaviour to make sure there are not any difference between those, regarding the requirements.

These methods are usually very time and resource consuming activities. The process is often undocumented, unrepeatable and unstructured, that's why creating tests limited by the ingenuity of the single developer. Furthermore the traditional test cases are static and hard to update, but the software under test is dynamically evolving. One other problem of the handcrafted test is, that they suffer from "pesticide paradox". The test are getting less effective during the testing process, because the tester writes them with the same method for mostly solved problems.

Model-based testing substitutes the traditional ad-hoc software testing methods with a well-defined process, which relies on behaviour models that describe the intended behaviour of the system and its environment. The subtasks of model-based testing are automatable, and set of test cases can be generated automatically from models and then executed on the tested software. The most difficult part of this process is the test case generation, which was solved many different ways in the last decade.

My research aims to create a new automated testing framework for software, based on state machine models. To do that, first I have to investigate the available solutions and techniques and related work. After summarising the conclusions, they can be used to design and implement a framework, that is able to generate test cases for softwares, modelled with state machines, which supports the most feasible state machine features regarding the UML semantics.

The tasks of my framework consists of creating the model of a given software, selecting test cases with a specific algorithm and formalising those generated test cases. So the resulted test cases can be used to run on software and verify its behaviour.

Proposed approach

First of all related work has to be examined. Similar solutions are available in the field of model-based testing, but the number of these solutions are limited. The basic problem has been solved

many times, but many of the solutions are not matured enough to be a perfect answer for testing real life softwares and do not support difficult structures.

The experiences from the previous research serve as a good starting point for the design of the framework. The most crucial questions to create this framework are the modelling language that is used to represent the abstract structure of the given software and the test generation algorithm.

The model has to have formal, hierarchically structured, modular, extensible metamodel, which can be transformed easily to an UML like metamodel. This is important, because we want to support test generation from state machines, which have an UML state machine like semantic. Supporting guards, actions and events natively in the model is also required.

Choosing a suitable test generation algorithm is also a challenging task. The used algorithm defines the functional and non-functional properties of the resulted framework, that's why the available solutions needs to be studied exhaustively. It has to generate a test suite, that is able to test all the needed functionalities of the given software with an acceptable theoretical complexity and execution performance.

At the implementation phase it is necessary to choose tools, which are easy to integrate. The best option for such an application is the Java ecosystem and the related tools, more accurately the Eclipse toolchain. Eclipse Modeling Framework offers the basic tools for model driven engineering. Possibly its metamodel, model editors, and code generation facility can be utilised to build the framework.

Scaling can be a big problem at test generation, that's why it is important to pay attention to this topic. The usage of variables at the model increases the state space, while the speed decreases. Maybe this could be a bottleneck, so monitoring and other measurements need to be applied to achieve the previously defined goals.

Chapter 1

Model-based testing

The idea of model-based testing originates from the 70's, and now it has an extensive literature, terminology and a commonly accepted taxonomy [**?**]. This section introduces the concept of this variant of software testing through a concrete process (Figure 1.1).

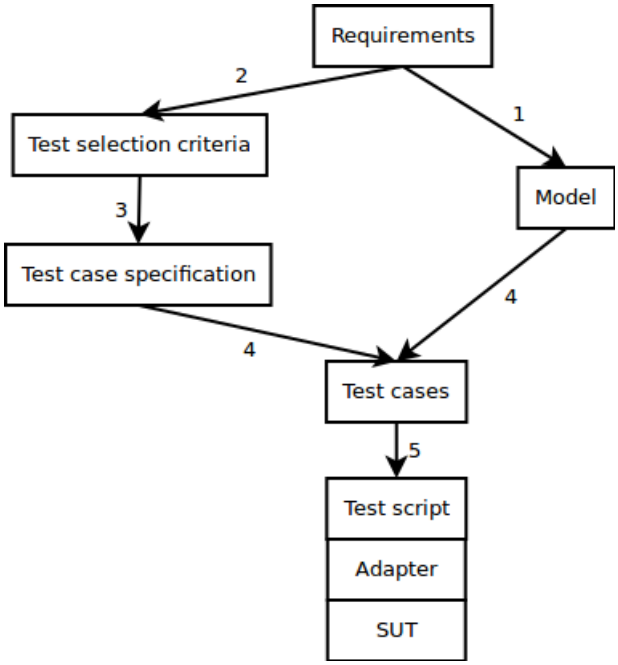

Figure 1.1. Model-based testing process

1. From informal requirements or created specifications a model can be built. The model is an abstract representation of the *system under test (SUT)*. It uses encapsulation for information reduction, because it has to be more simple, than the original system to achieve an easier modifying and maintaining [**?**]. During a model-based software development the model can be used for many other tasks too, as it serves analysing, synthesising and documenting the SUT as well.

2. Test selection criteria decide how the test cases are chosen, which point of view is important by testing. These selected criteria will control the whole test generation process.

3. Criteria are transformed into *test case specifications*. These test case specifications are the formalised versions of the criteria.

4. After creating the model and the test case specifications set of *test cases* is generated from the model regarding all the specifications. One of the biggest challenges is to create the test cases. A simple test case consists of a pair of input parameters and expected outputs. Finite set of test cases forms a *test suite*. The difficulty comes from the need to satisfy the test case specifications and create a minimised set of test cases.

5. A successfully generated test suite can be executed on the SUT. For the execution a *test script* can be used, which executes the test cases.

   The generated test cases are strongly linked to the abstract test model, therefore an *adaptor* component is needed, which is often part of the test script. The adaptor adapts the test inputs to the SUT. For example if the input of a method is an XML document containing an integer value, the adaptor has to transform the test case's test inputs to XML.

   The test script also contains usually a *test oracle*, that checks the test output difference from the expected output.

1.1 Taxonomy

Utting, Pretschner and Legeard investigated the currently available MBT solutions and defined (Figure 1.2) a taxonomy which concentrates to three major properties of model-based testing. The three dimensions of their taxonomy are the modelling specification, test generation and test execution.

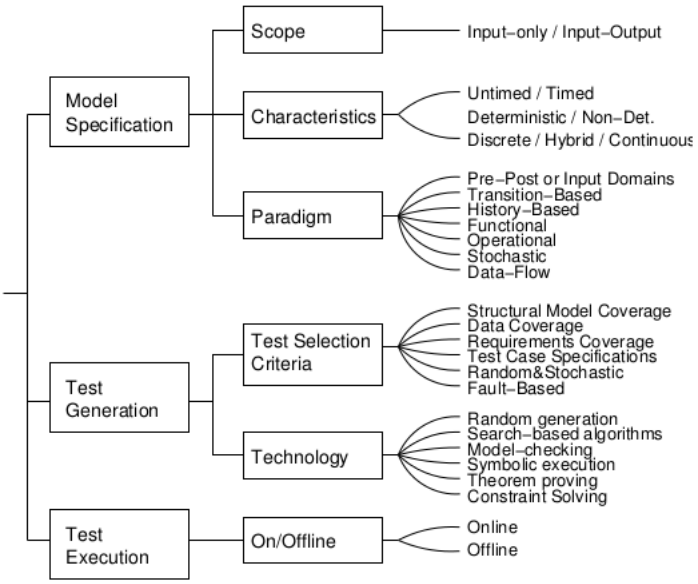

Figure 1.2. Model-based testing taxonomy [**?**]

Model scope The scope of the modelling is a binary decision. The model either specify *just the test input* or *the input-output pairs* for the SUT. The first case is less useful, because the test script can not check the SUT's output and that's why it is impossible to create an oracle that way.

Model characteristics The SUT assigns the main characteristics of the model. It depends on the SUT's timing properties (*timed* / *untimed*), determinism (*deterministic* / *non-deterministic*) and dynamics (*discrete* / *continuous* / *hybrid*).

Model paradigm The third dimension is the paradigm that is used to describe the model. *State-based notation* means, that set of variables defines the model, which represents the internal state of the system and there are some operations that modify those variables. Usually these operations given by preconditions and postconditions. By *transition-based notation* the model focuses on the transition between the state of the system. Finite state machines are examples of this paradigm. *History-based notations* model the allowable traces of its behaviour over time. By *functional notation* collection of mathematical functions model the system. *Operational notations* describe the model as a set of executable processes running parallel. Petri nets are good forms of this notation. *Stochastic notations* describe the model by a probabilistic model, so it is rather suitable to model the environment than the SUT itself. An example can be the Markov chains for this type of model paradigm. The last paradigm is the *data-flow notation*, where the main concept is the concentration to the data, rather than the control flow, example can be the often used Matlab Simulink model.

Test selection criteria Test selection criteria control the test case generation. *Structural model coverage criteria* aim to cover a part of the model, for example nodes and arcs of the transition-based model. The nodes of such a model represent the states of the system, and the arcs represent the transitions respectively. The basic idea of *data coverage criteria* is to split the data space to equivalence classes and choose values from them. *Requirements based coverage criteria* are linked to the informal requirements of the SUT and it applies the coverage to the requirements. *Ad-hoc test case specifications* guides by the test case specifications. *Random and stochastic criteria* are useful rather to model the environment and applicable to use with a stochastic model. *Fault-based criteria* can be very efficient, because it concentrates to error finding in the SUT.

Test generation technology One of the most important thing that defines the test case generation is the chosen technology. The easiest one to implement is the *random generation*, more difficult are the *search-based algorithms* where graph algorithms and other search algorithms are used to perform a walk on the model. *Model checking* can also be used for test case generation, where the model checker searches for a counter-example, which becomes a test case. *Symbolic execution* means analysing the software to determine what inputs cause each part of a program to execute. This method guided by test case specification to reach a goal, and test inputs become inputs which produce different outputs. *Deductive theorem proving* is similar to model checking, but the model checker is replaced with a theorem prover. *Constraint solving* is useful for selecting data values from complex data domains.

Test execution The tests can run either *online* or *offline* on the SUT. During an online test, the test generator can respond to the SUT's actual output for example with an different test case sequence. By an offline test generation test cases are generated strictly before the execution.

The testing can be started by an automatic execution or manually, that triggers the user directly.

1.2 Process

1.2.1 Modelling

The first step of the model based testing process is to create a suitable model, from which a test suite can be generated. As we earlier saw by the taxonomy section [**?**], all the identified model paradigms used in model based testing belong to some kind of behaviour modelling notation. This is not a surprise, because a data or functional model can not be utilised so effectively by software testing. Each model paradigm concentrates on a different aspect of the behaviour.

There is a plethora of technologies for modelling behaviour, and one of the most frequently used are the extended finite state machine (EFSM) and all of its variations. These variations mostly use transition based notation, but they can combine it with other modelling paradigms as well. The second most popular modelling language is the UML language, which is an enhanced version of EFSMs. Other modelling languages are used in the field of MBT too, but mostly these tools made for a specific purpose.

As EFSMs or at least their variations serve as basic modelling notation for the most available model based testing tools, that's why we have to investigate them properly. The basic parts of the UML language will be described here as well.

Extended finite state machines

A *finite state machine* is a 6-tuple $\langle S, I, A, R, \Delta, T \rangle$, where

S : set of finite states,

$I \subset S$: set of initial states,

A : finite alphabet of input symbols,

R : set of possible outputs,

$\Delta \subset S \times A$: set of possible input relations,

T : is a transition relation function $f : \Delta \to S \times R$

The semantic of this model is the following. When $T(s, a) = (s', r)$, the state machine is receiving an input $a \in A$, when in state $s \in S$, assuming $(s, a) \in \Delta$, then the system moves to the new state $s' \in S$, and outputs $r \in R$. A possible $(s'', a') \notin \Delta$ is interpreted as an input symbol, that is not allowed in that state.

An *extended finite state machine* differs from a simple finite state machine in terms of the states defined differently. The states of an extended state machine has the form $S = D_0 \times D_1 \times \ldots D_n$, where D_0 is the set of control states, and $D_{i=1}^n$ is the domain of state variables x_i, that are assigned to each states.

UML state machines

UML state machines or UML state charts are improved versions of the mathematical concept of finite state machines expressed with the OMG's Unified Modeling Language. The original state machine model suffer greatly by the state and transition explosion problem, because the complexity of these models tend to grow faster as the modelled system. UML state machines solved this problem by eliminating the common parts of these system and sharing the common behaviour across the states.

The behind idea behind the notation is, that an entity or each of its sub-entities is always in exactly one of the possible states and there are well-defined conditional transitions between these states. There are two kinds of state machine, which can be used to define behaviour of model elements or describe protocol usage.

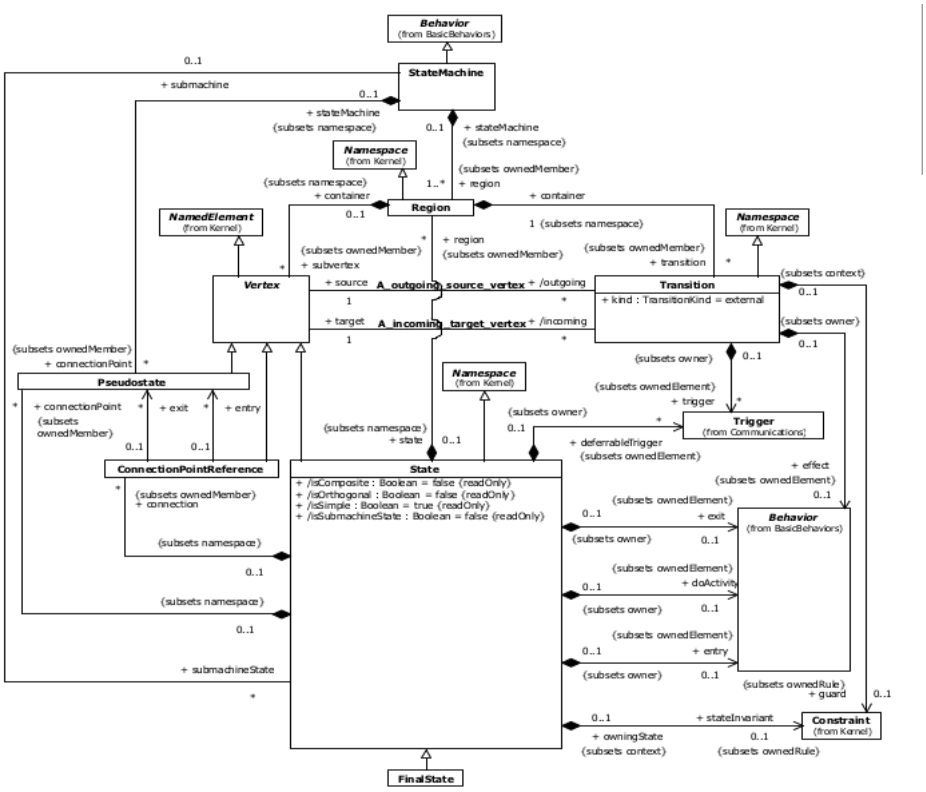

Figure 1.3. Metamodel of UML state machine

UML state machines are similar to traditional state machines, but they also have differences. For example UML state charts introduce new features over traditional finite machines such as hierarchically nested regions, orthogonal regions, entry/exit actions, internal transitions and transition execution sequences. The main concepts of this notation are discussed separately.

States are the phases of the system's history. For example if the history can be separated into two phases, then there are two states.

Extended states represents the complete condition of the system. This interpretation means usually states extended with system variables.

Transitions happens when a state switched to another.

Actions executed when an event dispatched and the system responds by performing them.

Events can be everything, that affects the system, and causes state change.

Guards are boolean expressions described with extended state variables and event parameters. They can affect the system's behaviour by enabling or disabling transitions.

Hierarchically nested regions means that if a system is in a substate then it is also in the same time in all the substate's superstates.

Orthogonal regions are regions, which are in 'OR' relation.

Entry/exit actions are actions which are dispatched upon entry to a state or exit from it.

Internal transitions do not cause state transitions, but only some internal actions to execute and the actual state stays the same.

Transition execution sequence describes an execution sequence of actions to do upon event dispatching. First the guard of the transition evaluates. Then the exit actions of the source state configuration will be executed. Then come the actions associated with the transition. Finally the entry actions of the target state configuration will be executed.

1.2.2 Test planning

Planning tests involves two steps considering the model based test generation process. At first the test selection criteria are chosen, which will be formalised into a test case specification later on.

The main goal of the test selection criteria is to guide the automatic test selection by the test case generation. A good criteria fulfils the previously defined testing policy and testing strategy, that were specified for the system [**?**]. Testing policies give rules for testing, while strategies are high-level guidelines.

Major tasks of test planning consist

- determining the scope of the testing and identifying its objectives.
- determining the test approach (techniques and coverage).
- implementing testing policy and the strategy.
- determining the required resources.

- scheduling the testing process.
- determining exit criteria such as coverage criteria.

The required output of the test selection criteria formalisation is the test case specification. This specification have to be fully formalised, so that a test generator is capable of generating test cases based on this formalisation and the software model.

1.2.3 Test generation

Investigating test case generation algorithms is important, because it has a strong impact on the effectiveness of software testing [**?**] [**?**]. That's why this topic is under activate research and resulted different approaches.

Adaptive random testing (ART)

Random testing is based on the idea, that the inputs have to spread across the domain of the input parameters to find failure causing inputs. There are five method in the field of ART:

- From a randomly generated input set, next candidate is chosen by a selected criterion.
- Next input parameter is chosen by exclusion: the randomly generated input parameter has to be outside of previously executed regions (exclusion regions).
- One other approach uses the information about previously executed input parameters, to divide the input domain into partitions. Next input parameter will be chosen from a new partition.
- The next input parameter can be chosen by dynamically adjusted test profiles.
- Distribution metrics can also help to find the next input parameter to achieve dispersion on the input domain.

Search based software testing (SBST)

In the last few decades there has been an exhausting research in the field of using graph theory at model-based testing. These techniques belong to search-based test generation algorithms.

One of the most used algorithms refers to the *Chinese Postman Problem* [**?**]. Given that it is impossible to cross each edge once in an undirected graph during a graph walk, in other words it does not have an Eulerian tour. What is the minimal amount of re-crossing we need to create a walk that uses each edge? The solution is to duplicate the shortest edges between the vertices having odd degree. This process is called "Eulerizing" the graph.

The *New York Street Sweeper Problem* is a variant of the previous graph theory problem. It applies to directed graphs, and arcs need to duplicate to reach, that each nodes have out-degree minus in-degree equal zero. In model-based testing one can use this idea, by creating a transition-based model, which can be represented as a graph. The vertices are the states of the SUT and the edges are the callable methods. A generated Eulerian tour gives a full transition-based structural model coverage.

The previous algorithms give full transition-based coverage, but not pair-wise coverage. The following algorithm named *de Bruijn sequences* creates every combination of the methods. First create a dual graph of the original graph, then eulerize the dual graph (by duplicating arcs to balance node polarities). Create an Eulerian tour, noting the names of the passed nodes.

Dill, Ho, Horowitz and Yang constructed worked on the *limited sub-tour problem* where the test case sequences can not be longer, than a specified upper limit. There is no optimal solution for that problem, but there are some heuristics. For example if an upper limit was set, the current sub-tour has to end and a new sub-tour has to start from that node.

Other approaches using a fitness function to find input parameters that maximises the achievement of test goals, while minimising testing costs.

Model checking

This is a traditional model based testing test case generation technique, where a model checker is used to generate test cases. The test criteria are defined as counter examples for the model checker. There are three main approaches in this topic, which are influenced by the chosen modelling notation [**?**]:

- **Axiomatic approaches** Axiomatic foundations of MBT are based on some form of logic calculus. The logic formula has to be transformed into disjunctive normal form (DNF), and this form has to be solved with a higher-order logical theorem prover or the problem has to be transformed into solving finite state machines.
- **Finite state machine approaches** The model is formalised with a Mealy machine, where inputs and outputs are paired on each transition. Test case generation is driven by some test selection criteria.
- **Labelled transition system approaches** This is a common formalism for describing operational semantics of process algebra. There are two common techniques generating test cases (input/output conformance and interface automata), which describe the conformance of the SUT. These techniques do not define test selection strategies, they have to be combined with coverage criteria as seen by FSMs.

Symbolic execution

Symbolic execution is a program analysis technique that analyses a program's code to automatically generate test cases from it. It belongs to white box testing, because the inner structure of the SUT is known during the test.

Symbolic execution uses symbolic values, instead of concrete values, as program inputs. During the symbolic execution the state of the program is represented with *symbolic values* of program variables at that point, a *path constraint* created by symbolic values, and a *program counter*. The path constraint is a Boolean formula, that has to be satisfied to reach that point on the path. At each branch point the path constraint is updated with constraints of the inputs. If the path constraint becomes unsatisfiable, the path can not be continued. If the the path constraint stays satisfiable, then all solution for the Boolean formula can be an input for a given test case.

There are numerous tools which proves the usefulness of this technique, but there are three main problem that limits the effectiveness of this method by real world programs.

- **Path explosion** The most real world program have a huge number of computational path. The execution of each path can be mean an unacceptable overhead. Solutions for this problem can be using the specification of the parts that affect the symbolic execution or avoiding some branch, which are irrelevant to the test data criteria.
- **Path divergence** Programs usually implemented in a mixture of different programming languages. The symbolic execution of such a complex infrastructure is almost impossible. The unavailability of these paths leads to path divergence, and some paths may not be found during the symbolic execution. Possible solution can be to replace these paths with a model during the test generation.
- **Complex constraints** Solving Boolean formulas involves using constraint solvers during the symbolic execution. There are some formula that, which can not be solved with the today available tools. These formulas can be simplified by replacing solvable sub formulas with concrete values.

Combinatorial testing

In combinatorial testing samples of input parameters have to be chosen, that cover a prescribed subset of combinations of the elements to be tested. Usually sample consists all t-way combination of possible input parameters, this method is called *combinatorial interaction testing* (CIT). The inputs can be described with a covering array:

$$CA = (N; t, k, v),$$

where N represents sample size, t is called strength, k are the factors and v are the possible symbols. So CA is an $N * k$ array on v symbols such that every $N * t$ sub-array contains all

t-tuples from the v symbols at least once. Finding an appropriate coverage array is possible using heuristics.

Combinatorial testing can be used if the domains of the input parameters are known.

1.2.4 Test execution

Test execution includes several steps, because the abstraction level of the generated test cases differ from the SUT. Therefore a previously mentioned adaptor component is needed that bridges between the two component. The concrete execution is done by a component named test script, which includes a test oracle that determines, if the test were run successfully or not.

The tasks of the execution are the followings:

- Execute the complete test suite or individual test cases with test scripts.
- Log the outcome of the execution and report the identities and versions of the SUT and the testing tools.
- Compare the results with the expectations using oracles.
- Report the differences between the actual and the expected results.
- Repeat the execution with the same configuration to prove the correctness of a previously failed test case. When we just re-execute a test case that called *confirmation testing*, but we have to check that a fix does not introduce new defects (*regression testing*).

Chapter 2

Related work

Model-based testing is a mature idea, and it has an extensive literature. Nevertheless the number of the available, useful tools is less than we can expect that. To really take advantage of model-based testing, reliable tools and automation support are required. A usable model-based testing tool has to help in the whole testing process. That means creating and verifying the model, generating test cases, constructing test scripts, adaptors and oracles.

Utting, Pretschner and Legeard [**?**] defined MBT as testing that relies on models specifying the intended behaviour of the SUT. In reality that would mean restricting MBT to black-box testing, where we can only generate abstract test cases from the behaviour model. That's why Shafique and Labiche defined MBT as a support of software testing activities from a model of the SUT behaviour. I follow this point of view in this paper.

Shafique and Labiche [**?**] collected the available tools that rely on state-based models and created a systematic review considering the previously and newly defined criteria. The defined criteria summarise the essential parts of model based testing softwares and can be used in this thesis as well, to learn from these tools, what did they well or what kind of feature do they miss.

First I will describe the applied review protocol, then I will explain in more detail the usage and the available features of some popular testing tool. Finally I will collect and summarise all the data, that is needed to start designing a useful model based testing framework.

Model-flow criteria This criterion details the used test selection criteria by the actual tool. These test selection criteria refer to a chosen coverage options, which can be state, transition, transition-pair, sneak path, all-paths and scenario criteria. The first five are well-known, scenario criteria means, that the test should follow user defined test sequence to pass.

Script-flow criteria Some MBT tool extends the semantics of the original EFSM notation to modify the SUT behaviour more precisely. They can use some script language or pre/-post conditions to specify the behaviour more further. These mechanisms provide some more lower-level criteria that the tools can consider, for example interface, statement, decision/branch, condition, modified-condition/decision and atomic condition coverage.

Data criteria This criterion refers to the selection of input values when creating concrete test cases from abstract test cases. The options are one-value, all-values, boundary-values and pair-wise values. By one-value only one concrete test case will be generated for an abstract test case, by all-value all concrete test case will be generated for an abstract test case. Boundary-value means selecting values from a specific range.

Requirement criteria It is a binary decision whether a tool supports checking of requirement's satisfaction or not. Requirements are linked to a specific part of the model (e.g. transition, state), to a third-party tool or to other requirement sources.

Scaffolding criteria Scaffolding means generating part of a required code. Fully support refers to scaffolding out all needed part of the process, partially support means only a few of them.

2.1 GraphWalker

The first investigated tool was GraphWalker [**?**], which can create online and offline tests from finite state machines, extended finite state machines or from both of them. The framework is written in Java, the related tools belong to Java world as well. Maven is used to run the tests, TestNG to describe the test cases.

The input model has to be in GraphML format, which is an easy-to-use, highly extendable XML extension for describing graphs. The creators of this software think that UML is too complex, and its functionality is not necessary by software testing, that's why they chose this format. Recommended tool to create GraphML is yED, which is a graphical graph editing software.

After designing the model, test stubs, adaptors and oracles will be generated. The adaptor has to be filled with the linking logic with the SUT. While running the tests GraphWalker can use different methods to walk on the state space. For example A* search, shortest path, random path, all permutation. The tests will stop when a certain stop criterion has been satisfied. The stop criteria can be state coverage, transition coverage, requirement coverage and time limit.

2.2 PyModel

PyModel [**?**][**?**] is an open-source MBT testing framework written in Python. It consists three main tools:

pma - PyModel Analyzer It validates the model program and creates FSM from it.

pmg - PyModel Graphics It generates a graph representation of the FSM.

pmt - PyModel Tester It creates online and offline test cases and executes them.

PyModel's input model are given by FSM specification or by code named model program. The methods will be the transitions in the FSM, states are the defined attributes. It is possible to

combine different models in a test. Scenarios supported as well, so user can guide the tests with a given test case sequence. There are two test coverage criteria, state-based and transition-based coverage.

2.3 Conformiq

Conformiq Designer is one of the most famous, industrial model based testing tool. It is available as a plugin for Eclipse, and in a form of a standalone testing framework. Conformiq generates test cases using specific criteria, verifies the test model, executes the test suite and offers coverage reports.

Conformiq is a complete framework, that support the whole model based testing process. Seeing the success of this software, the design of the software has to be investigated. The MBT process using Conformiq is identical to the original high level MBT method as it can be see on Figure 2.1.

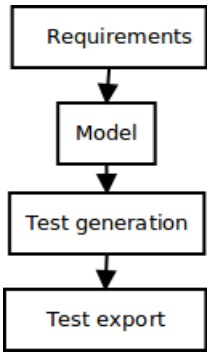

Figure 2.1. MBT process in Conformiq

1. First step is specifying requirements. Conformiq support a huge amount of industrial requirements modelling tool (eg. IBM Rational, Rhapsody, Sparx Systems Enterprise, ArchitectHP Quality Center, IBM RequisitePro, DOORS), but it contains an own editor too. The defined requirements are traceable through the whole software testing process.

2. Based on the requirements one has to create the model of the SUT. It can be done with the Conformiq Designer internal model editor using its QML language. The language consists three parts: system block diagrams, which describes the interface of the model (inbound and outbound ports); UML statecharts and Java like action language.

3. After the modelling phase abstract test cases can be generated. The generation starts with transforming the model to an intermediate Lisp model, that is used during the symbolic execution, which generates the use cases. The user is able to see coverage statistics and a traceability matrix based on the generated test cases.

4. Abstract test cases have to be exported with so called scripting backend which creates concrete test cases for the SUT.

2.4 GOTCHA

GOTCHA is a framework that consist of two main components. The first one generates test cases from FSM models, while the others transforms abstract test cases into concrete test cases written in Java and then executes them on the SUT.

The model is described with GDL (GOTCHA Definition Language), that contains states, state variables, actions, expected results and guards. Abstract test cases are generated into XML format. Concrete test cases are executed on the SUT using an adapter, that can be written with the help of some helper class. The concrete method mapping is made by an XML file, that maps to specific SUT methods.

2.5 ParTeG

Partition Test Generator is an open source Eclipse plugin, that can generate test cases from UML models annotated with OCL guards. It traverse the graph representing the UML state machine and each path corresponds to a test case.

The used test case generation algorithm is the following:

1. A selected coverage criterion is transformed into model specific test goals.
2. Each test goal references a concrete element of the model.
3. From each of these element a path to the model's initial state represents a test case given by the corresponding transitions.
4. Backwards on the path, each guard becomes a constraint on the inputs, which will be the initial input parameters in the end.

2.6 Conclusions

After investigating five widely used MBT tools, we can draw some consequences. From our point of view the most important parts of their features are the used model notation and the test case (TC) generation methods, because these are the most crucial part of the design. I summarised the collected information in the Table [**?**].

- Creators of these tools either try to use an UML like model or FSM (EFSM). FSM models are low level representations of the SUT, so implementing search based algorithms and graph traversal algorithms are relatively easy.

  When engineers choose to use model with UML with graph intermediate model, they can not support complex UML state chart elements, such as orthogonal regions, because these features are hard to integrate into a graph representation.

Name of the tool	Model	Intermediate model	TC generation method
GraphWalker	FSM	graph (GraphML)	search based, combinatorial, random
PyModel	FSM + Python	graph	search based, random
Conformiq	QML	Lisp (CQλ)	symbolic execution
GOTCHA	EFSM	graph	BFS, DFS
ParTeG	UML + OCL	graph	DFS, symbolic execution

Table 2.1. Summary of examined MBT tools

- The intermediate model is just always some kind of graph representations, because the test case generation algorithms are the easiest to implement using graph models (search based test case generation, coverage criteria).
- Scaffolding solutions of the tools are incomplete. Fully automatic generation of test adaptors, oracles are seldom supported. These features make the testing tool more useful, because they accelerate the testing process.
- Regression tests are not supported. When an actual error is found, then the SUT should be tested against the generated test suite and this process should be supported by the testing tool.
- Only a few tools have an integrated solution to create models. Handling models correctly is an essential feature of model based testing tools, because an integrated model editor improves the testing process greatly. Testing is an iterative process, so contextual switching between model editor and testing tool results an overhead.
- Input models are not verified. Model based testing the same as other testing methods can only find discrepancies regarding their source. If the test model is not correct, then the tests will be ineffective. That's why that is also important to verify the input model, and to help the testing process it can be built into the testing framework.
- The tools implement different coverage criteria, but even the most general state and transition coverage are not fully supported by each of the tools. More difficult criteria are avoided, for example transition pair, sneak path, all path and scenario coverage.

  Transition pair coverage may be avoided because the few added value compared to a full transition coverage. Another reason can be, that depending on the actual model and test case generation algorithm this criterion can be hard to implement properly.

  Sneak path means a path that contains an accepted method, that should not be accepted. By a fully specified model each possible transition is represented, so that sneak path criterion is not applicable.

  Usage of scenario coverage results reasonable smaller test suite, then the other test selection criteria, but a transition coverage may replace this criterion.
- Script-flow criteria are rarely used techniques. Only a few of them supports guards, but even those do not report on their coverage. Simple criteria as model flow criteria are not

so effective at finding faults, whereas complex criteria like script flow criteria help find different kind of faults. On the other hand complex criteria are also significantly expensive in terms of theoretical complexity.

- Requirement traceability are ignored by just all the available tools. This feature make the tools more useful, but does not effect the ability to find more errors. Tracing requirements is rather used by software validation.

Chapter 3

Design

In this chapter I will present the design phase of a new model based testing framework, that tries to take into consideration of the conclusions of the investigated available testing tools. This framework is need to be complete regarding the whole MBT process, and has to help in all phases of the testing. The different phases will be discussed separately as at the model based testing process specification (Section [**?**]).

1. The first step is the creation of the model. There are many possibility to choose from and we want to have a transition based notation that represents some kind of state machine. State machine notations have different level of expression and come with different amount of features. Generally the more feature a modelling language has, the more hard to generate a good quality test suite from it. That's why we need to find a notation that has a suitable level of expressiveness and it is easy to integrate into a complete testing process.

   Earlier we saw that the lowest level of state machine notation is some kind of FSM like notation. However FSMs lacks of many features and a real world software is hard to model with it. Actions, guards, events are not even parts of the improved EFSM notation, so we need to find something more expressive.

   Many tools have an UML like notation, but they either can not fully take advantage of the many features of this modelling language or simply avoid their usage. That is not surprising because UML was not designed for testing purposes. UML has just all the features, that are needed to describe the behaviour of a real life software, but some of its feature are hard to utilise during the test generation process. Moreover it lacks some important feature, that a test model has to bear with.

   It would be ideal if the test model would express the output of the state machine, because determining the expected output would be trivial by the test generation process. However an UML like semantics and syntax would be easy to adopt by the test engineers, because UML state machines are well known in the industry field and most of its features are easy to use and self-describing. Unfortunately the UML modelling language has lot of implementation and they differ in terms of integrability.

For these issues noted above I think a solution can be the Eclipse Modeling Framework. It is a modelling framework that is built especially for creating tools based on structured models. As the most important advantages of using EMF help with the implementation, it will be discussed in more details in Chapter cha:implementation.

EMF models are easy to use, extendable models, that have an UML like syntax. These models are customisable for the actual needs, and suitable meta-model can be built with the help of the EMF platform. PLCspecif is complete solution that has exactly these previously defined features.

3.0.1 PLCspecif

PLCspecif is a modelling language intended to be a formal, modular, hierarchical behaviour specification method for describing PLC programs. It was created as part of a doctoral programme by Dániel Darvas of the Budapest University of Technology and Economics (BUTE) and the European Organization for Nuclear Research (CERN).

The abstract syntax of the PLCspecif formalism was designed as an EMF metamodel, therefore the figures are following the original EMF denotation. Here I will describe only the concerned parts of the modelling language as the complete feature set goes far beyond this thesis.

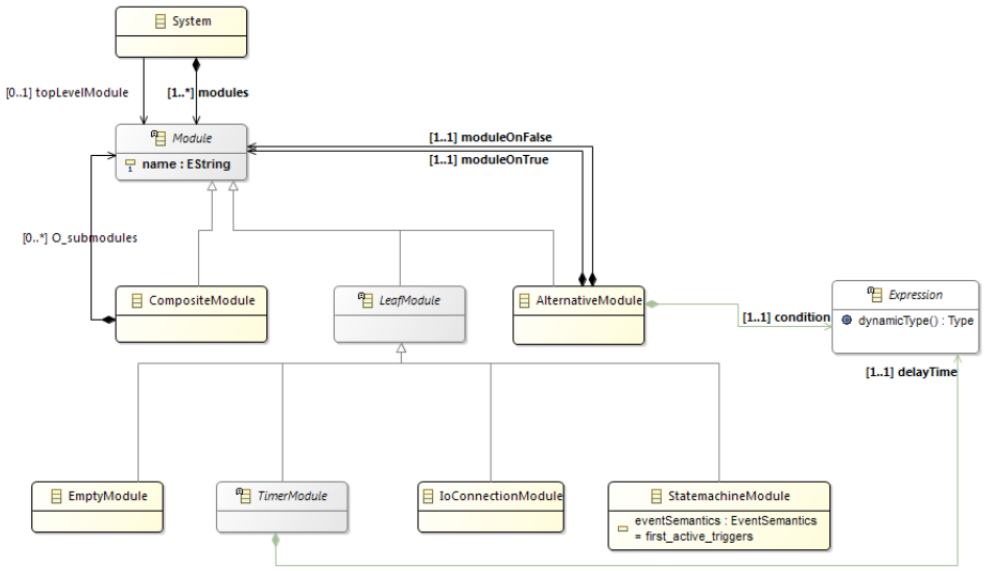

Figure 3.1. Module structure of PLCspecif

The specification organised into modules (Figure 3.2), which are either represent a behaviour of concrete module (`LeafModule`) or they are composite modules containing a set of submodules (`CompositeModule`).

`System` is a top-level container that can contain `modules` from which one module represents the `topLevelModule`. There are four different module type:

- `StatemachineModule` represents an UML-like state machine.

- `IoConnectionModule` defined by connections between input and output variables.
- `TimerModule` describes a PLC timer in the system.
- `EmptyModule` is a module without any state machine or IO connection.

From these module types we are interested especially in the state machine notation. As shown on Figure 3.3 the metamodel is similar to UML state machine's metamodel described previously (Subsection [**?**]).

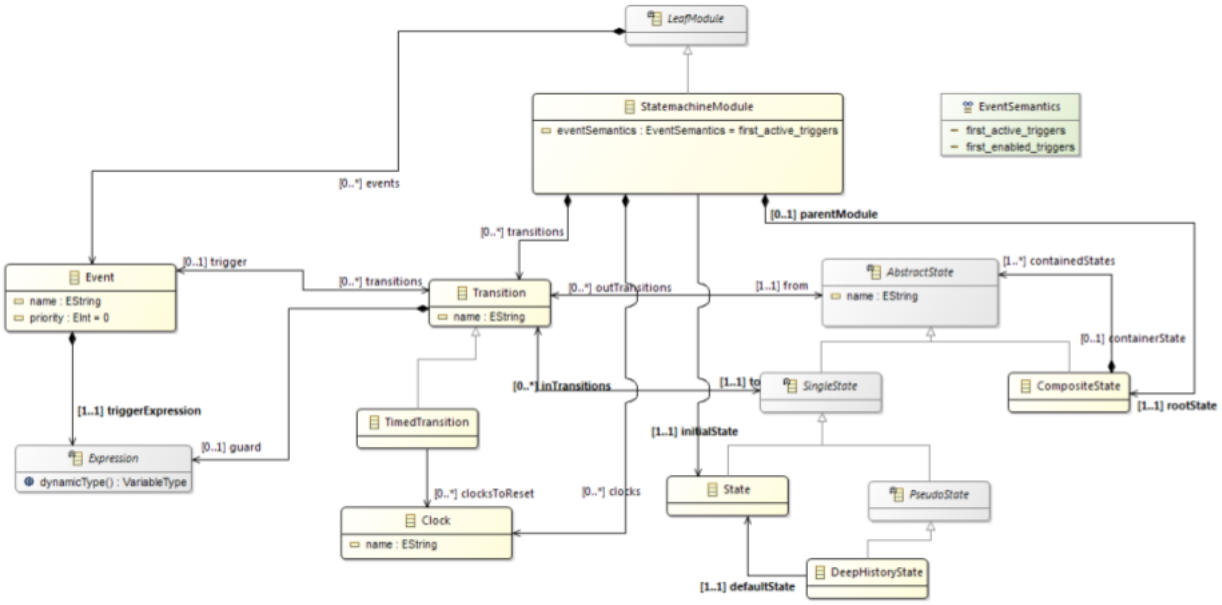

Figure 3.2. Structure of `StatemachineModule`

On the other hand PLCspecif state machines have some differences from the UML notation:

- There is a root state, that recursively contains all of states.
- There are pseudo states (`DeepHistoryState`), which can save a state configuration for its container state.
- There are `TimedTransitions`, which are transitions having time-related conditions.
- With `Clocks` it is possible to define synchronous stopwatches, which can measure the elapsed time since last reset.
- Parallel regions are not allowed.
- Initial state can not be defined for composite states.
- At every moment, exactly one atomic state can be active.

Beside these differences PLCspecif has a bigger difference, which can improve the usability of a test model. Modules can handle input and output variables and can define their outputs using `VariableDefinitionExpression` objects (Figure [**?**]). PLCspecif offers a wide range of possibilities to define an `Expression`,

e.g. `SwitchCaseTable`, `DnfExpression`, `Contant`, `UnaryOperation`, `BinaryOperation`, `NaryOperation`. In our case the output definition of a single state machine can be a switch case table, where conditions checks whether the state machine is in a particular state, and the values are the possible values given by a variable or constant.

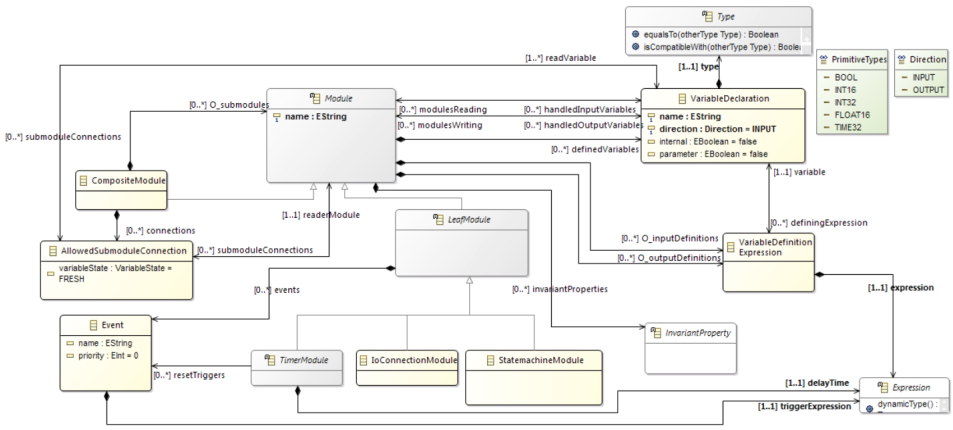

Figure 3.3. Variables in PLCspecif

We can see, that PLCspecif has some advantage over traditional UML modelling language in the field of model based testing:

- EMF Ecore is a reference implementation of OMG's Essential Meta-Object Facility, that's why syntax should conform with the original UML denotation.
- PLCspecif is rather a subset of the UML state machine language, and so it is more simple.
- PLCspecif is able to express natively the outputs of a state machine, which can be utilised greatly by the test case generation.
- As PLCspecif is based on the EMF Ecore metamodel engineers can take advantage of the entire EMF ecosystem and tooling by the whole MBT process.
- UML language has only an informally given semantics, while PLCspecif has a complete formal behaviour specification.

2. The next step of model based testing is the test planning.

   The scope of a generated test suite is always is referred to the corresponding state machine. In reality following the Law of Demeter and decoupling, that would mean one class or two classes.

   Another important question to discuss is the specification of chosen test selection criteria. We saw at the conclusions of the investigation of related works (Chapter [**?**]), that even the most general and simplest criteria are not fully supported by all the available tools. They either implement complex criteria with a less useful, simple model, or they work with an expressive model and do not support the criteria completely.

At this point I tried to find a golden mean between the different approaches. I chose to implement the basic structural model coverage criteria (full state and transition coverage) on a model with moderate level of expressiveness. Formalisation of this criteria is mostly trivial and left to the implementation chapter ([**?**]).

3. The third step is the test case generation. After examining the available tools we saw, that generally simple test case generation algorithms work fine on simple SUT representations, but as we increase the level of expressiveness so gets also the execution more slower. To generate test suites from complex models we need something more powerful.

   Traditional MBT test generation technologies involve model checking, deductive theorem proving and constraint solving. These methods all embed and utilise some powerful mechanism to generate test cases using complex criteria and models. To get the most of these methods usually an intermediate model is used, that maps the original test model to an applicable form and can be executed by e.g. a model checker. That's why our test model should be easily transformable to the intermediate model's notation.

3.0.2 Alloy

Alloy is a formal modelling language based on first-order logic to define structures, complex structural constraints and behaviour. Alloy can be utilised with a tool, called Alloy Analyzer to automate the verification process.

The language has been developed on MIT and the first prototype was finished in 1997 in the form of a limited object modelling language. Later the features, performance and scalability have been improved.

Alloy is a declarative language, so that it describes the behaviour without giving the precise execution mechanism. The language was influenced by the Z notation, which is a formal specification language used describing and modelling computer programs and systems. In contrary of Z, Alloy was designed for automatic analysis.

OCL (Object Constraint Language) with UML are often used by MBT tools for expressing SUT behaviour. UML semantics can be imitated by an Alloy model, but Alloy can do even more. As OCL similar to Alloy, but the latter has a more conventional syntax and simpler semantics. So Alloy can combine the features of the two OMG standards and can serve as the input model of a model based testing tool.

Alloy Analyzer transforms problems into SAT formulas to solve them. The solver was inspired by model checkers, but it is implemented as a constraint solver, performing verification within a bounded scope. In constraint programming relations between variables are noted in the form of constraints, that will be solved by giving a value to each variable so that the solution is consistent. If the constraints are inconsistent, then the problem is said to be unsatisfiable.

Alloy version 4 ships in the form of a self-contained JAR file, which includes a variety of supported SAT solver, the standard Alloy library, tutorial examples and an extensive API, that's why it is easy to incorporate into a custom solution.

To summarise the statements above, Alloy has the following benefits:

- Alloy are largely compatible with the UML notation. Transforming a model given by any state machine notation should not be a problem.
- Alloy is a declarative modelling language, which has the same advantages as any other declarative language. Using a declarative language often results reusable code, and smaller codebase as the imperative versions.
- Alloy Analyzer has a convenient API, which are easy to integrate in a tool written in Java.

4. The last step of the testing process is the test execution. The available MBT tools seems to avoid the support of this step. This is somewhat surprising, because the real theoretical and technical difficulties are solved in the previous steps.

   Test scripts, adaptors and oracles can be generated from the prepared test suite and the SUT. EMF code generation facility are a perfect tool to help by this step as well, as by the test case generation.

Chapter 4

Implementation

At the end of the design phase it is clearly visible, that the investigated technologies and tools what advantages have. To adopt the chosen technologies I chose to use the Java programming language. For model driven engineering the Eclipse platform serves the best tools, that's why I used Eclipse Modeling Tools to implement the application.

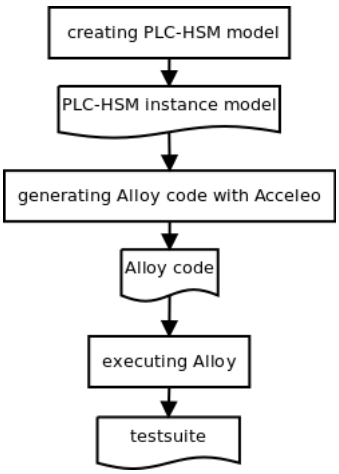

Figure 4.1. Architecture of the test generator framework

The test case generation process consists of the following steps (see Figure 4.1):

1. First the instance model have to created with the default PLC-HSM model editor generated by the Eclipse Modeling Framework. The model can have all the PLC-HSM model features except the timed transitions.

2. The next step is to create Alloy code, that can produce the test cases. The required informations can be extracted from the previously created PLC-HSM model, and so the desired Alloy code can be generated automatically. This generation was solved with Acceleo, which is a model to text transforming tool as part of the Eclipse Modeling Tools.

   The generated Alloy code guarantees state and transition coverages. To create the Alloy code, we need to know the name of states, transitions, their relationship, the guards and the

initial state of the SUT. From these information will be the necessary Alloy signatures and predicates generated.

3. The generated Alloy code can be executed with Alloy Analyzer to get the test suite with all the test cases.

The generated Alloy code will be demonstrated with an example (see Figure 4.2). The static part of the generated Alloy code can be see on Listing 4.1.

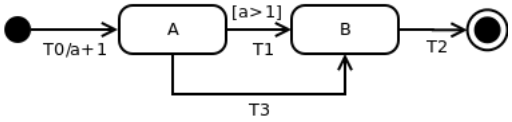

Figure 4.2. Example state machine with guard

The basic state machine element's (system, states, transitions) are between line number 4-7. The next section (line number 9-18.) describes the structure of a basic test case. One test case consists of several steps. The only given fact (line number 20-27.) defines the connection between a test case and the state machine. The predicate `inheritSystem` is a utility method, that can be used to inherit extended state variables from previous states. Predicates `transition_coverage` and `state_coverage` define transition and state coverage criteria accordingly. These predicates can be executed using the `run` statement used in line number 38.

Listing 4.1. Test suite generator Alloy code

```
1  module psm_statecoverage
2  open util/integer
3
4  abstract sig System {}
5  abstract sig State {system: one System}
6  abstract sig Transition {from, to: one State}
7  one sig Initial, End extends State {}
8
9  sig TestCase { firstStep: Step }
10 sig Step {
11         from, to: State,
12         via: Transition,
13         nextStep: lone Step
14 } {
15         via.from = from
16         via.to = to
17 }
18 fun steps (tc:TestCase): set Step { tc.firstStep.*nextStep }
19
20 fact {
21         all s:Step, tc:TestCase | s in tc.firstStep.*nextStep
22         all tc:TestCase | tc.firstStep.from = Initial
23         all t:Transition | one s:Step | s.via = t
24         all curr:Step, next:curr.nextStep | next.from = curr.to
25         all sys:System | some s:State | sys = s.system
26         all s:State | some t:Transition | t.from = s or t.to = s
27 }
28
29 pred inheritSystem(s1, s2: System) { s1 = s2 }
```

```
30
31 /***** GENERATED CODE START *****/
32 ...
33 /***** GENERATED CODE END *****/
34
35 pred transition_coverage() { some tc:TestCase | steps[tc].via = Transition }
36 pred state_coverage() { some tc:TestCase | all s:State | s in steps[tc].from + steps[tc
       ].to }
37
38 run state_coverage for 10 but exactly 1 TestCase
```

The dynamic part of the Alloy code, generated from the instance model can be see on Listing 4.2. The code starts with the initialization of the SUT. The structure was defined in a signature, while the initial state of the SUT needs to define in a predicate. The rest of the code describes the other parts of the state machine: the states (`A`, `B`), the transitions (`T0`, `T1`, `T2`, `T3`), the events (`E0`) and the guards (`G0`).

Listing 4.2. Dynamically generated Alloy codes

```
1  sig S extends System { a: Int }
2  pred initSystem(s:System) { s.a = 0 }
3
4  one sig A, B extends State {}
5  lone sig T0 extends Transition {}{
6          from = Initial
7          to = A
8          initSystem[from.system]
9          E0[from.system, to.system]
10 }
11 lone sig T1 extends Transition {}{
12         from = A
13         to = B
14         inheritSystem[from.system, to.system]
15         G0[from.system]
16 }
17 lone sig T2 extends Transition {}{
18         from = B
19         to = End
20         inheritSystem[from.system, to.system]
21 }
22 lone sig T3 extends Transition {}{
23         from = A
24         to = B
25         inheritSystem[from.system, to.system]
26 }
27 pred E0(s1, s2: System) { s2.a = add[s1.a, 1] }
28 pred G0(s: System) { s.a > 1 }
```

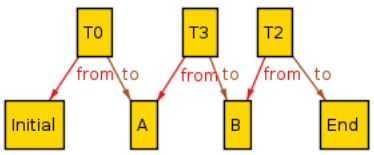

Figure 4.3. Example test case generated from state machine in Figure 4.2

The above Alloy code generates test cases with state coverage guaranteed and the resulted test case is on Figure 4.3 considering the previously defined state machine. As we can see the transition `T1`, having an unsatisfiable guard, is left out from the test case, and the generated test case satisfies all the requirements.

Chapter 5

Scaling and measurements

After each implementation iteration I measured the performance of the created framework, and continued the development using the results of these measurements. As we previously saw the heart of the framework is the SAT solver, which is also the most time consuming part of the system. So the best way to improve the speed of the execution is to improve the underlying Alloy program.

I created a testing tool to measure the execution of the different Alloy programs. This testing tool can be configured to compare the execution of different Alloy programs, with different execution strategy. The execution strategy can mean different SAT solvers, and other solver configurations as well. As an input for this testing tool I designed an example FSM, that represents a simplified stopwatch behaviour (Figure 5.1). This FSM is ideal for testing purposes, because on the test suite of this stopwatch full state and transition coverage can be achieved and therefore both of the implemented algorithms can be tested.

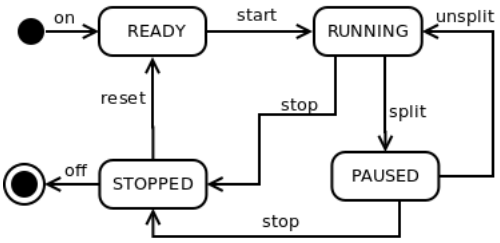

Figure 5.1. Stopwatch FSM for testing

The tests has been running on the following configuration (see the specification in Table 5.1).

Hardware specification
CPU: 2.7GHz dual-core Intel Core i5 processor with 3MB shared L3 cache
RAM: 8GB 1866MHz LPDDR3 RAM
Storage: 128GB PCIe-based flash storage

Table 5.1. Measurement architecture

5.1 Alloy settings

During the first measurement I experimented with the available solvers supported by Alloy and their solver specific configurations.

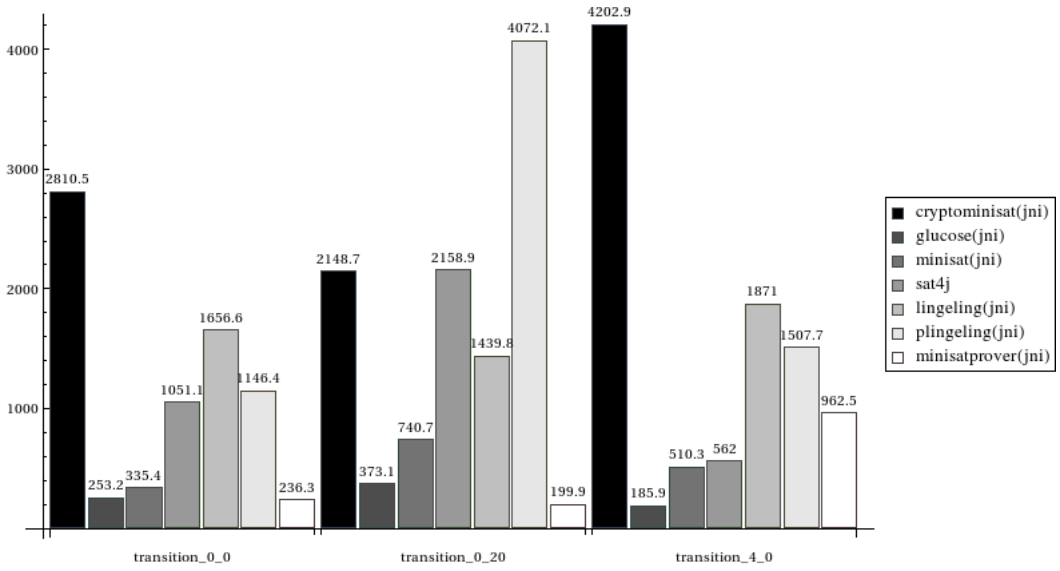

Figure 5.2. Adjusting Alloy settings

5.2 Optimalizations

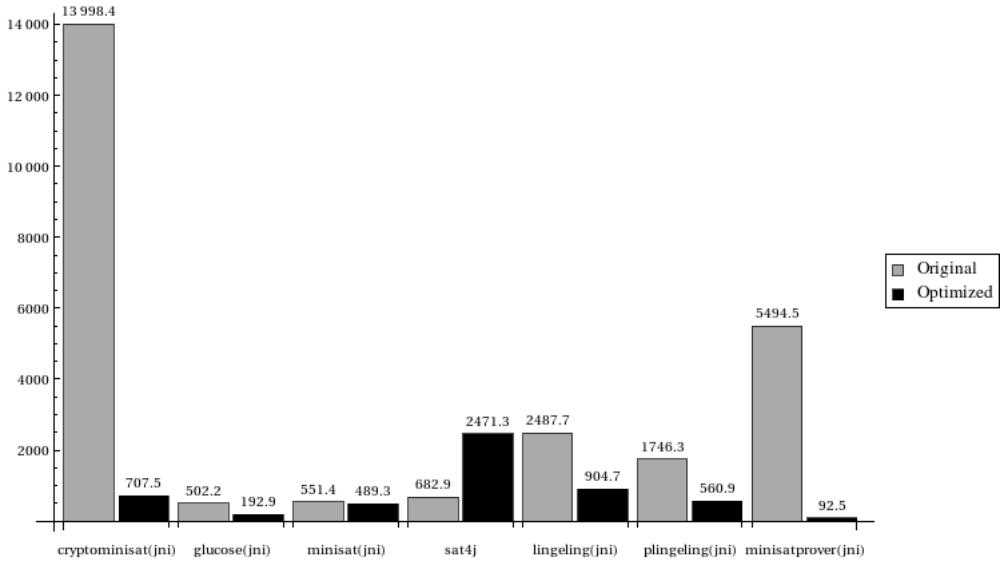

Figure 5.3. Optimalisation results

5.3 Scalability

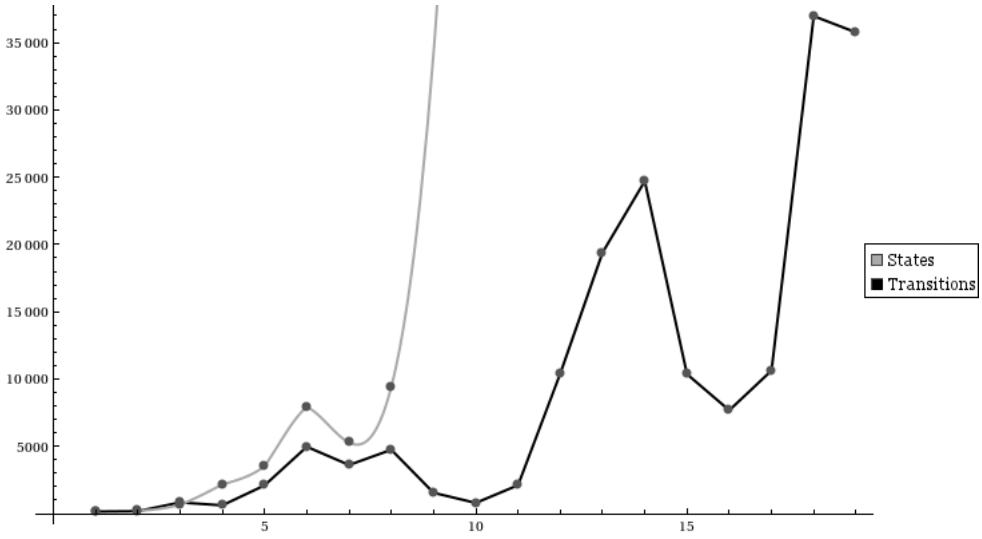

Figure 5.4. Scalability results

Chapter 6

Summary and further development

...

List of Figures

List of Tables

Bibliography

Appendix

The test generator and the additional tools belongs to Apache License 2.0 licence.

Web:
`https://bit.ly/testgeneration`
Git URL:
`git@github.com:thesnapdragon/msc-thesis.git`
Licence:
Apache License, Version 2.0