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 21, 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 21.

Unicsovics Milán György
hallgató

Kivonat

...

Abstract

...

Introduction

Problem and thesis statement

Software testing is important part of any software development process, because it is one 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, 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. The used algorithm defines also the functional and non-functional properties of the resulted framework.

At the implementation phase that 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.

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).

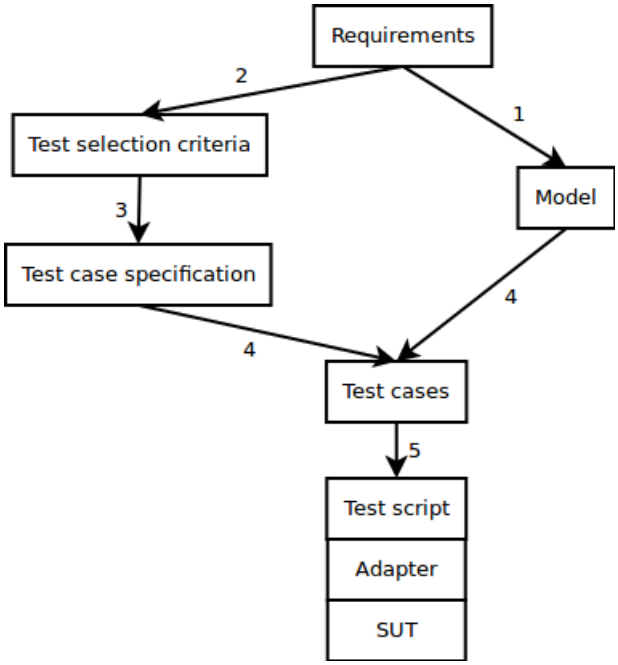

Figure 1.1. Model-based testing process

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 to information reduction, because it has to be more simple, than the original system to achieve an easier modifying, maintaining [**?**]. During a model-based software development it can be used for many other tasks too, as the model 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.

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

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.

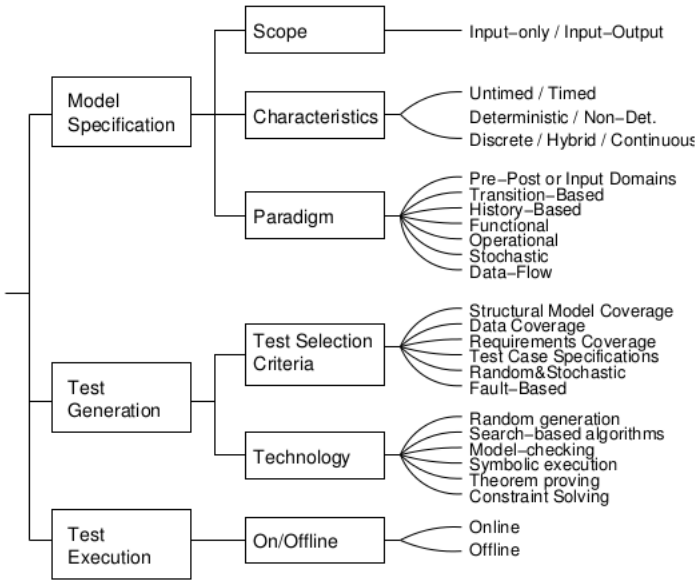

Figure 1.2. Model-based testing taxonomy [**?**]

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. By *transition-based notation* the model focuses on the transition between the state of the system. Finite state machines are examples for 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. *Stochastic notations* describe the model by a probabilistic model, as it is rather suitable to model the environment than the SUT itself. The last paradigm is the *data-flow notation*, where the main concept is the concentration to the data, rather than the control flow.

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 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

...

1.2.2 Test design

...

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. The available method should be studied, because the efficiency can be improved by combining these methods.

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 use the specification of the parts that affect the symbolic execution or avoid some branch, which are relevant 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.

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.

Model based testing (MBT)

The known test case generation techniques, that are used in model based testing are introduced here. The model based testing terminology and process was presented last semester in the final report of the previous project laboratory.

There are three main approaches by traditional model based testing:

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 cases can be generated using some coverage criteria. These criteria was discussed last semester.

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.

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.

Adaptive random testing (ART)

Random testing is based on 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:

1. From a randomly generated input set, next candidate is chosen by a selected criterion.
2. Next input parameter is chosen by exclusion: the randomly generated input parameter has to be outside of previously executed regions (exclusion regions).
3. This 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.
4. The next input parameter can be chosen by dynamically adjusted test profiles.
5. 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. 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 *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.

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.

1.2.4 Test execution

...

Chapter 2

Related work

Model-based testing is a mature idea, and it has an extensive literature. Nevertheless the number of the available 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. We 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.

Model-flow criteria This criterion details the state-based coverage options and applicable to state-based models, which belong to transition-based models. The coverage options 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 This criterion refers to interface (function), statement, decision/branch, condition, modified-condition/decision and atomic-condition. They can extend the finite state machine's mechanism. Interfaces refer to the functions which are called on the SUT, the others can serve as guards on transitions.

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).

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 test stub 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 are 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 and creates FSM from it.

pmg - PyModel Graphics It generates graphical representation of the FSM.

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

PyModel's test input has to be created by code. 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. 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 process as it can be seen on Figure 2.1.

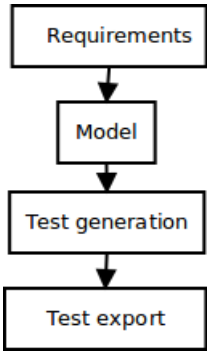

Figure 2.1. MBT process in Conformiq

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

...

2.5 ParTeG

...

2.6 Conclusions

After investigating five widely used MBT tools, we can draw some consequences.

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

Table 2.1. Summary of examined MBT tools

- The tools implement different coverage criteria, but mostly just the easiest ones (state-based and transition based version of structural model coverage criteria). More difficult criteria are avoided, for example data coverage, requirement-based and fault-based criteria and transition pairs coverage from structural coverage criteria.
- Script-flow criteria are rarely used techniques. Only a few tools support guards on transitions or use scripts for example to give control information of the SUT.
- Scaffolding solutions of the tools are incomplete. Fully automatic generation of test adaptors, oracles are seldom supported.
- Regression tests are not supported.
- 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 tools using UML model with graph intermediate model, they can not support complex UML state chart elements, such as orthogonal regions.
- 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).
- Important thing to note, that the most successful tools use symbolic execution and it can handle the most complex models.

Chapter 3

Design

Generating test cases from a specific model based on two step from the model-based testing process (see fourth step at Chapter 1), namely the modelling and test case formalization (test selection criteria, test case specification) phases. These steps will be discussed one by one, considering the previously defined requirements.

3.1 Modeling language

3.1.1 UML state machine

As the goal is to generate test cases from state machines, that having an UML state machine like semantics, investigating the UML standard [**?**] can not be missed in this thesis.

UML state machines or UML state charts are improved versions of the mathematical concept of finite state automatons expressed with the OMG's Unified Modeling Language. The idea behind this notation is, that an entity or each of its sub-entities is always in exactly one of a number of possible states and where there are well-defined conditional transitions between these states. UML state charts introduce new features over traditional finite automatons such as hierarchically nested regions, orthogonal regions and actions. There are two kinds of state machine, which can be used to define behaviour of model elements and describe protocol usage.

UML state machines are similar to traditional state machines, the main parts of them are the followings:

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.

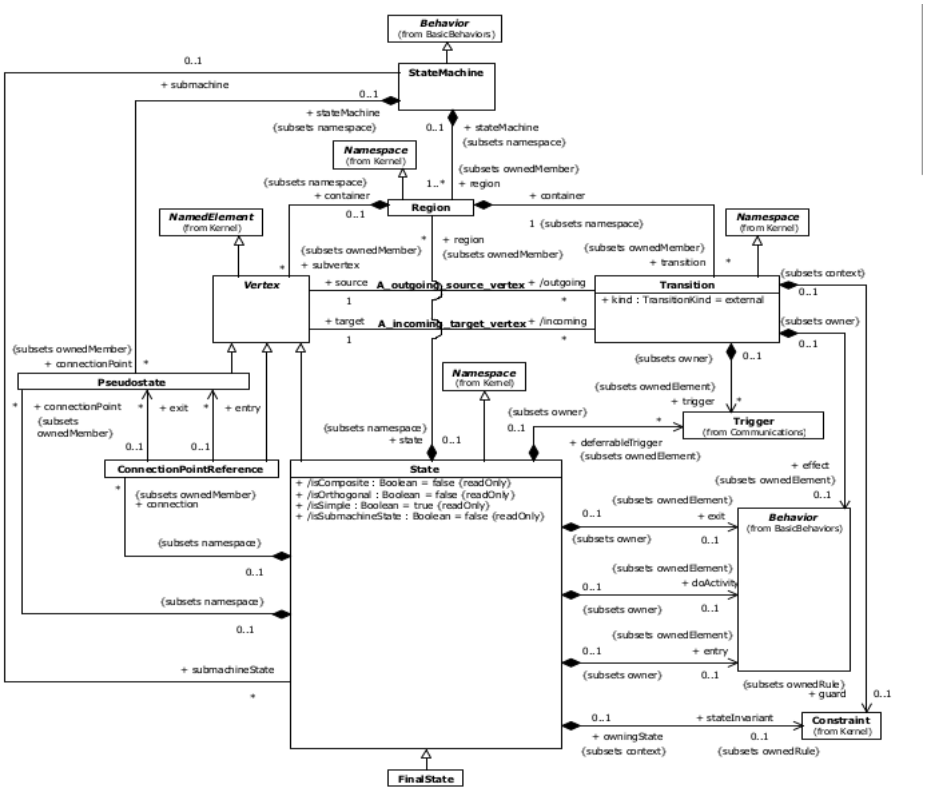

Figure 3.1. Metamodel of UML state machine

Events can be everything, that happens with 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.

3.1.2 PLC-HSM

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

The specification organized into modules (Figure **??**), which are either represent a behaviour of concrete module (`LeafModule`) or they are composite modules containing a set of submodules (`CompositeModule`). There are four different module type:

- `StatemachineModule` represents an UML-like state machine.
- `IoConnectionModule` defined by connections between input and output variables.

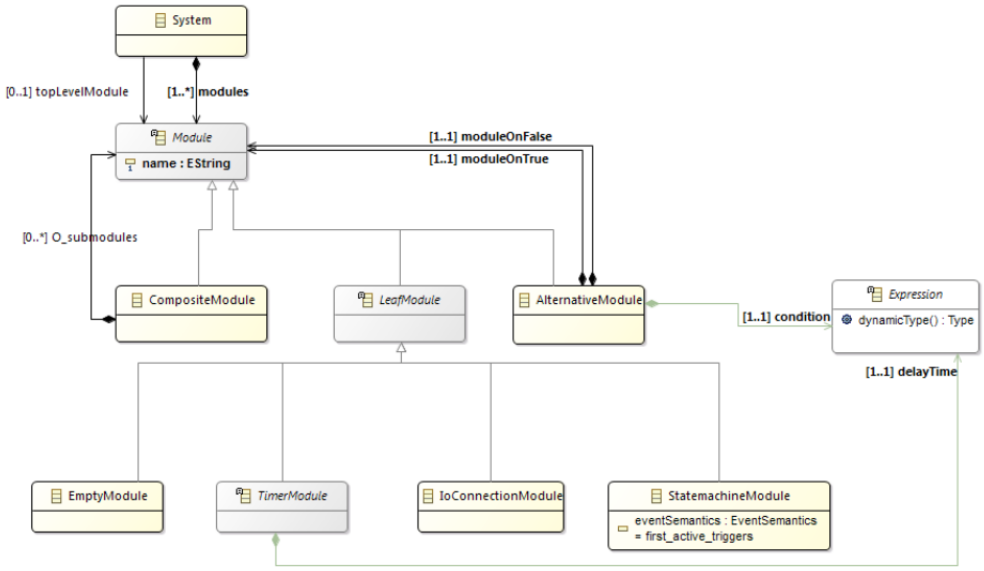

Figure 3.2. Module structure of PLC-HSM

- `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 **??** the metamodel is similar to UML state machine's metamodel described in the previous section.

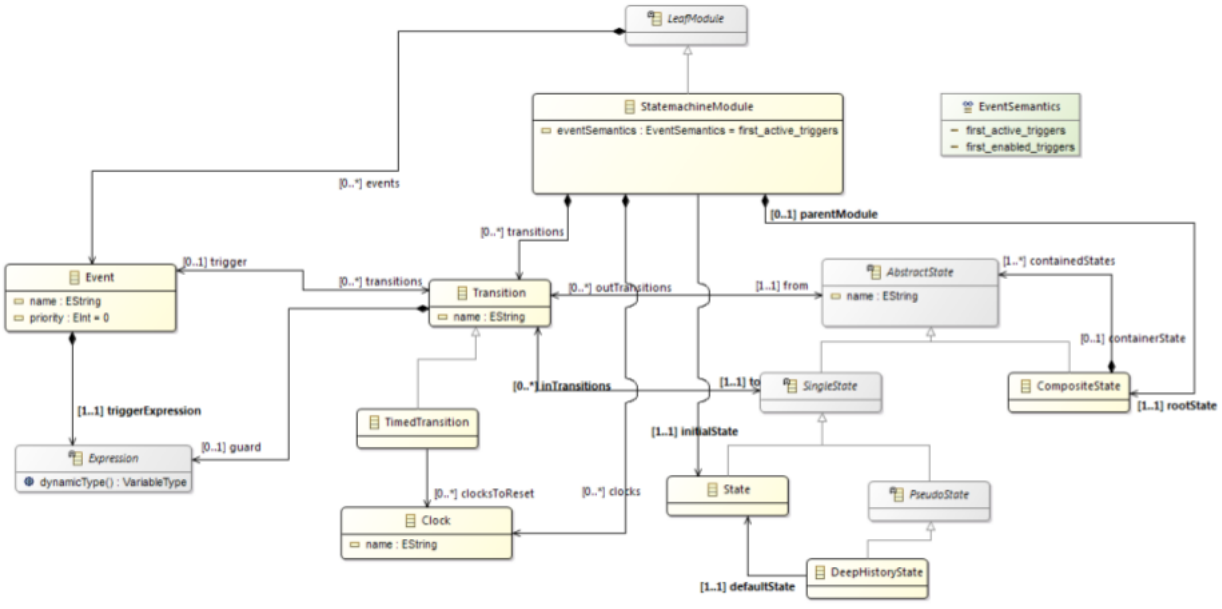

Figure 3.3. Structure of `StatemachineModule`

On the other hand PLC-HSM has some difference 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.

We can see, that PLC-HSM has some advantage over traditional UML modeling language:

- UML language has only an informally given semantics.
- Tools having UML state machine creating capabilities are not standardised.
- PLC-HSM is rather a subset of the UML state machine language, and so it is more simple.

3.2 Test generation algorithms

3.2.1 Alloy

Alloy is a formal modeling language to define structures. Alloy can be utilised with a tool, called Alloy Analyzer to automate the verification process. The tool transforms problems into SAT formulas to solve them. The solver was inspired by model checkers, but it is implemented as a solver, performing verification within a bounded scope.

The strength of this tool allows us to define our test generation goals with the Alloy language to generate the test cases. The test cases need to guarantee state-based and transition-based coverages.

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.

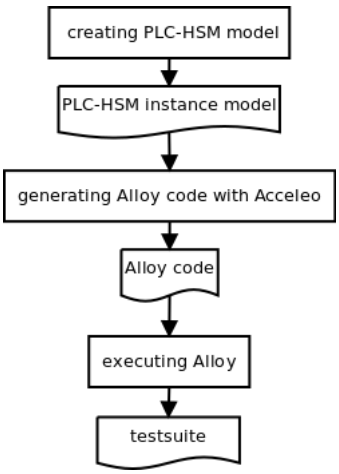

Figure 4.1. Architecture of the test generator framework

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

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 **??**). The static part of the generated Alloy code can be see on Listing **??**.

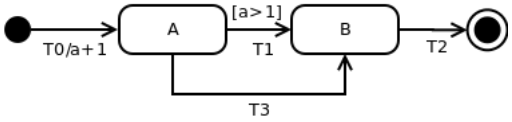

Figure 4.2. Example state machine with guard

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 **??**. 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 }
```

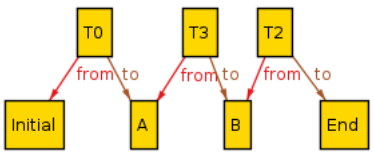

Figure 4.3. Example test case generated from state machine in Figure **??**

The above Alloy code generates test cases with state coverage guaranteed and the resulted test case is on Figure **??** 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 saw previously 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 **??**). 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.

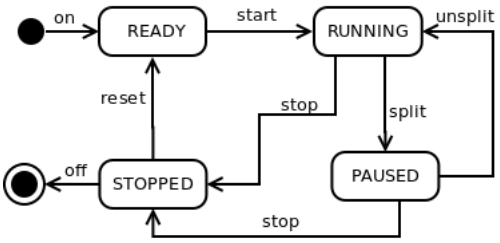

Figure 5.1. Stopwatch FSM for testing

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

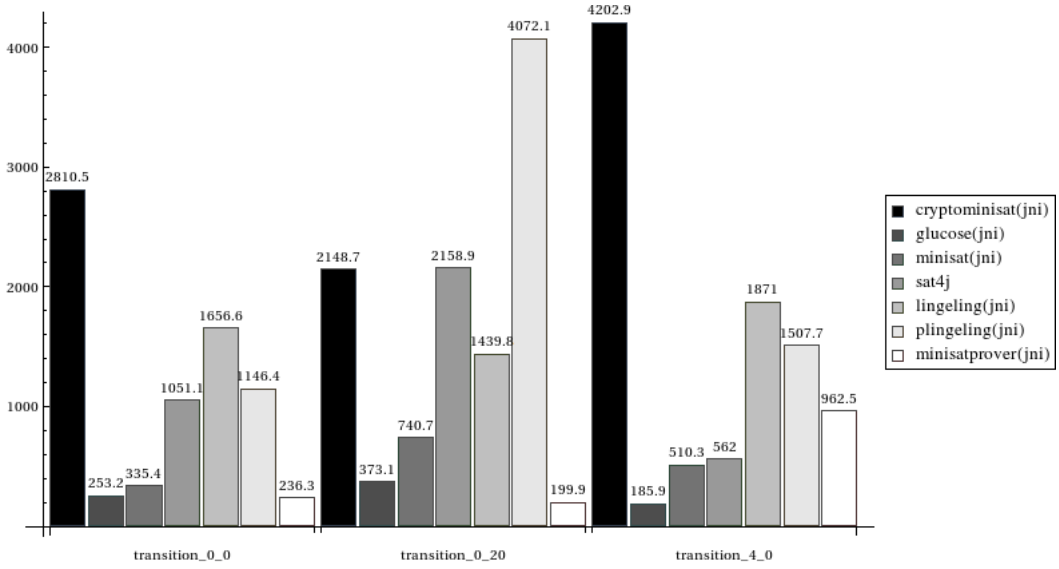

Figure 5.2. Adjusting Alloy settings

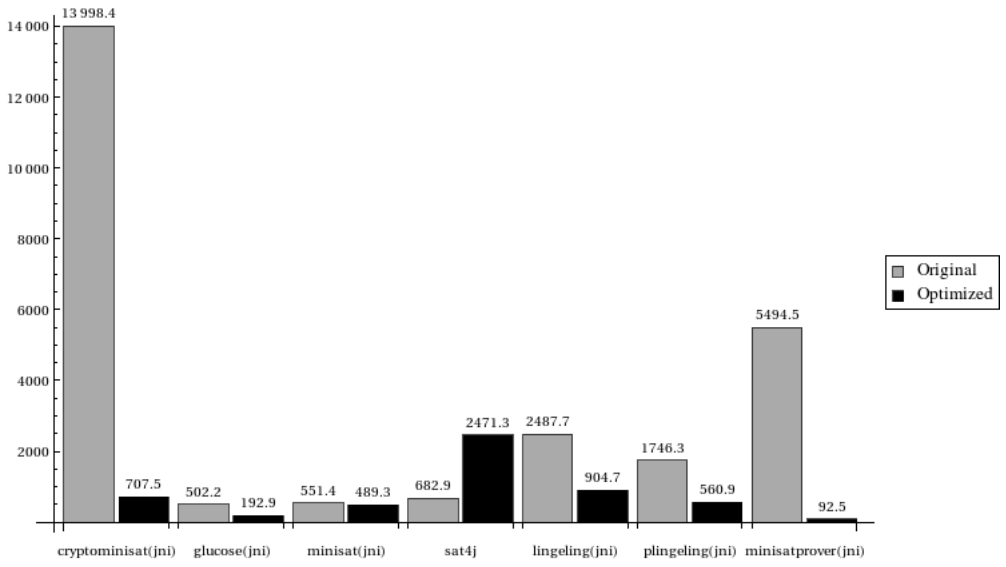

Figure 5.3. Optimalisation results

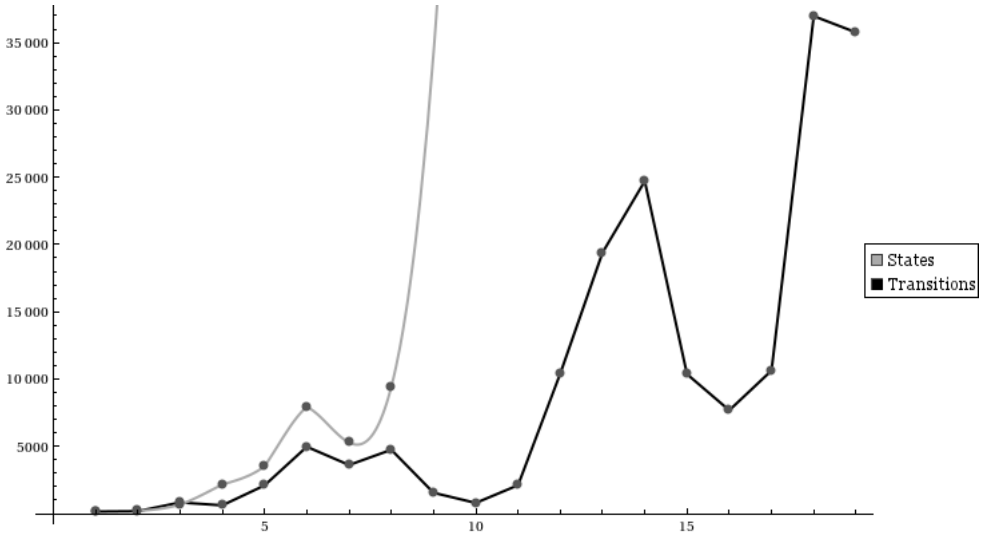

Figure 5.4. Scalability results

Chapter 6

Summary and further development

...

List of Figures

List of Tables

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