MŰEGYETEM 1782

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

December 17, 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. december 17.

Unicsovics Milán György
hallgató

Kivonat

A modell alapú tesztelés a szoftvertesztelés egy változata, ahol a szoftver viselkedésének verifikációja történik meg egy korábban definiált viselkedés modell alapján. A tesztelés ezen formája megoldásként tud szolgálni a tradícionális szoftvertesztelés régóta ismert és égető problémáira. Habár a modell alapú tesztelés alapgondolata a 70-es évekből ered és a szakirodalom mennyisége számottevő, jelentések azt mutatják, hogy az elérhető megoldások nem teljeskörűek és gyakran csak az eredeti probléma egyes részeire szolgálnak megoldással.

Diplomám célja, hogy bemutassam egy modell alapú tesztelő keretrendszer fejlesztésének teljes életciklusát. Az elkészített keretrendszernek a tesztelési folyamat minden fázisát támogatnia kell, hogy a rendszer képes legyen az adott szoftver egy teljes tesztkészletének generálására állapotgép alapú modellek alapján.

A tesztelési folyamat támogatásához megismertem a modell alapú tesztelés elméleti hátterét. Kutatásom fő célja volt, hogy a különböző tesztfázisok teendői azonosítva legyenek és hogy minden információ készen álljon a rendszer elkészítéséhez.

Az összegyűjtött tapasztalatok alapján a publikusan elérhető modell alapú tesztelő keretrendszerek megismerése jó alapként szolgálhat a készülő megoldás fejlesztéséhez, ezért összegeztem ezen eszközök előnyeit és hátrányait.

Ezután összegyűjtöttem a szükséges információkat és meghoztam a legfontosabb tervezői döntéseket, így elkezdődhetett a fejlesztés. A főbb architektúrális kérdések, a szükséges technológiák és eszközök meghatározása szintén a tervezési folyamat része volt.

Az implementációs fázisról szóló fejezetben részletesen bemutattam az elkészített rendszer képességeit, illetve annak belső működését. A tesztelés különböző lépései egy egyszerű példán keresztül ismertettem. A rendszer belső állapota és az átmenetileg elkészült eredmények bemutatása is ennek a példának a felhasználásával történtek meg.

Az elkészült keretrendszer képességeit mérési eredményekkel is alátámasztottam. A szoftver teljesítményét a fejlesztés alatt folyamatosan mértem, így a különböző iterációk eredményei összehasonlíthatóak.

Végül a mérési eredmények alapján kiértékeltem az elvégzett munkát és továbbfejlesztési lehetőségeket is meghatároztam.

Abstract

Model-based testing (MBT) is a variant of software testing, where the behaviour of the software is verified against a previously defined behaviour model. Verifying softwares, using this method, can solve the most crucial parts of traditional software testing and may also offer some other benefits. Although MBT is a mature idea and the field is well studied, reports show that available solutions are not fully complete and often targeted to solve only subparts of the original problem.

This thesis aims to present the full development lifecycle of a model-based testing framework that can help in all phases of the testing and that is able to generate test suites based on state machine like modelling notations.

To fully support the whole testing process I had been investigating thoroughly the background of model-based testing. Main goal of this research was to identify the primary tasks of the different testing phases and to collect all possible information that are needed to create this testing tool.

Using this knowledge I examined the related work. Comparison of available MBT tools can serve as a good starting point to develop a comprehensive solution, therefore summarising the experiences about these tools also important.

After gathering all the required information I made some design choices to start the development. Main architecture, the necessary technologies and tools have to be selected at the design phase as well.

The implementation chapter demonstrates the features of the testing framework and describes the internal behaviour in details. The different steps of the testing are exemplified by generating a test suite for a trivial software. States of the internal structures and intermediate results are showed also regarding this trivial example.

Results of the finished implementation are represented by measurements. I measured the performance of the framework during the development, thus the improvement after each iteration was quantifiable. According to these measurements the resulted software can be evaluated, proposing new features and room for future improvements.

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.

Testing is usually very time and resource consuming activity. The process is often undocumented, unrepeatable and unstructured, that's why creating tests is 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 tests 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 a 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, 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 software, modelled with state machines, which supports the most feasible state machine features.

The tasks of my framework consist 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 the 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 is 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 a real life software 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 design, 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 a feature set similar to an UML state machine like notation. 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 need 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. A good candidate for such an application can be 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 [30]. MBT can be defined as a software testing technique, where the software's intended behaviour is verified against a formerly constructed model. This chapter introduces the concept of this variant of software testing through a concrete process (Figure 1.1).

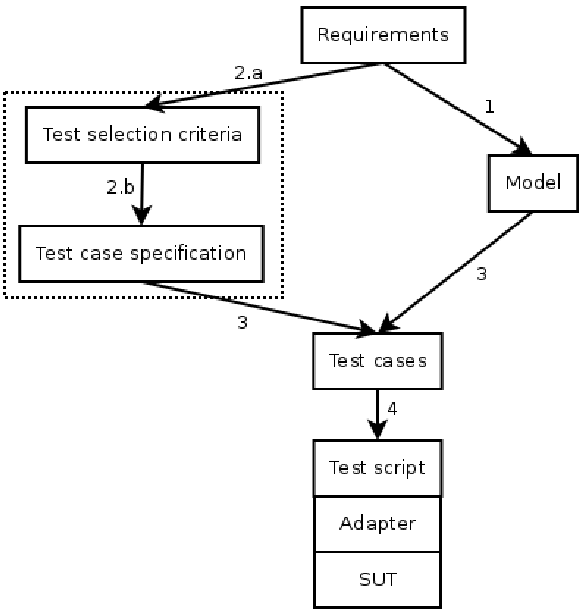

Figure 1.1. Model-based testing process

1. Modelling From informal requirements or previously defined 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 should be more simple, than the original system to achieve an easier modifying and maintaining [3]. During 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 planning *Test selection criteria* decide how the test cases are chosen, which point of view is important by testing. Later these selected criteria will control the whole test generation process. Criteria are transformed into *test case specifications*, which are the formalised versions of the criteria. These two steps are often treated separately, but they form a cohesive step of test planning, thus they will be discussed together in this thesis.

3. Test generation After creating the model and the test case specifications, set of *test cases* is generated automatically 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 to create a minimised set of test cases.

4. Test execution 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 *adapter* component is needed, which is often part of the test script. The adapter adapts the test inputs to the SUT. For example if the input of a method is an XML document containing an integer value, the adapter has to transform the test case's test inputs to XML.

The test script usually contains a *test oracle* that checks the difference between the test output and the expected output.

Utting, Pretschner and Legeard investigated the currently available MBT solutions and defined a taxonomy (see Figure 1.2), 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, which will be followed and expanded by the presentation of each stages of the testing process.

1.1 Modelling

The first step of the model-based testing process is to create a suitable model, from which a test suite can be generated. Model specifications has three dimensions considering the different MBT approaches.

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

Model characteristics The SUT determines 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 are given by preconditions and postconditions. By *transition-based notation* the model focuses on the transitions between the states 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.

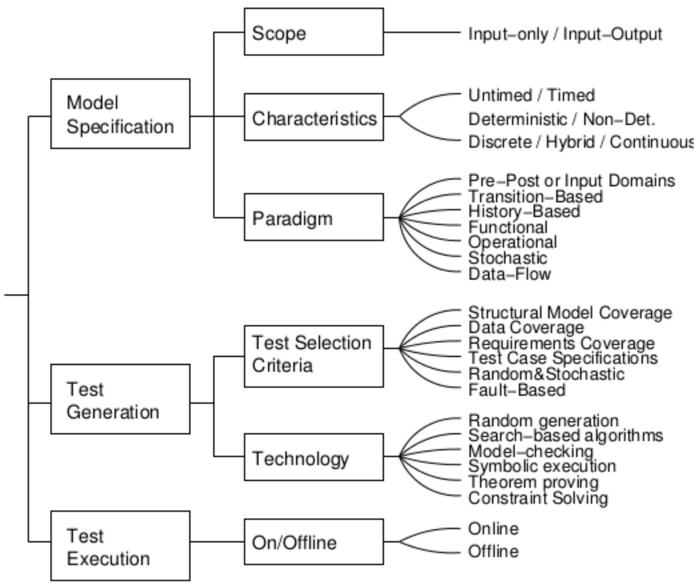

Figure 1.2. Model-based testing taxonomy [30]

As we saw by the taxonomy, 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 to 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 according to Shafique and Labiche [27] is the UML state machine language, which is an enhanced version of EFSMs. Other modelling languages are used in the field of MBT too, but mostly these tools are made for specific purposes.

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.

1.1.1 Extended finite state machines

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

$$
\begin{aligned}
S &: \text{set of finite states,}\\
I \subset S &: \text{set of initial states,}\\
A &: \text{finite alphabet of input symbols,}\\
R &: \text{set of possible outputs,}\\
\Delta \subset S \times A &: \text{set of possible input relations,}\\
T &: \text{is a transition relation function } f : \Delta \to S \times R
\end{aligned}
$$

The semantic of this model is the following. When $T(s, a) = (s', r)$, the state machine is receiving an input $a \in A$ 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 have 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.

1.1.2 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 [24]. The original FSM notations 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 extracting the common parts of these system and sharing the common behaviour across the states.

The 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 define the behaviour of model elements or describe protocol usage.

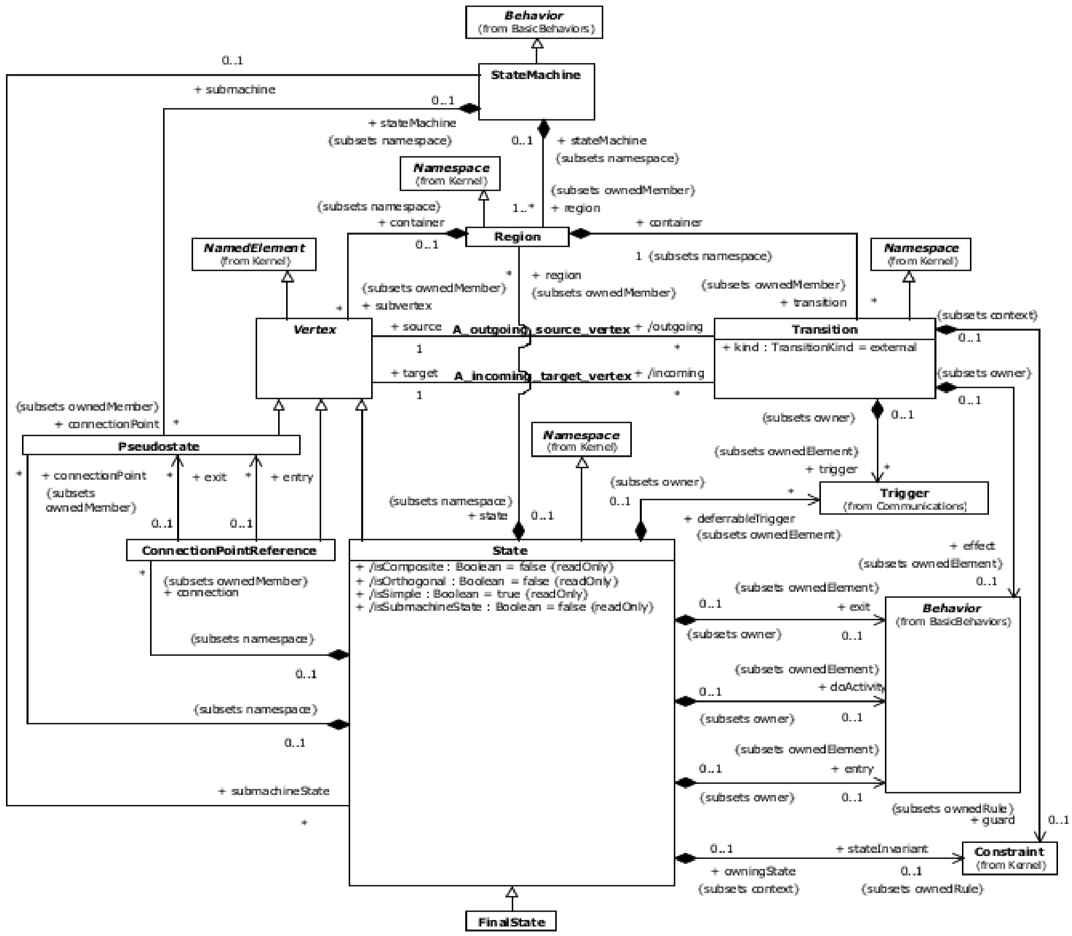

Figure 1.3. Metamodel of UML state machine [24]

UML state machines are similar to FSMs, 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 represent the complete condition of the system. Usually this is implemented with states that are extended with system variables.

Transitions happen when a state switched to another.

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

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

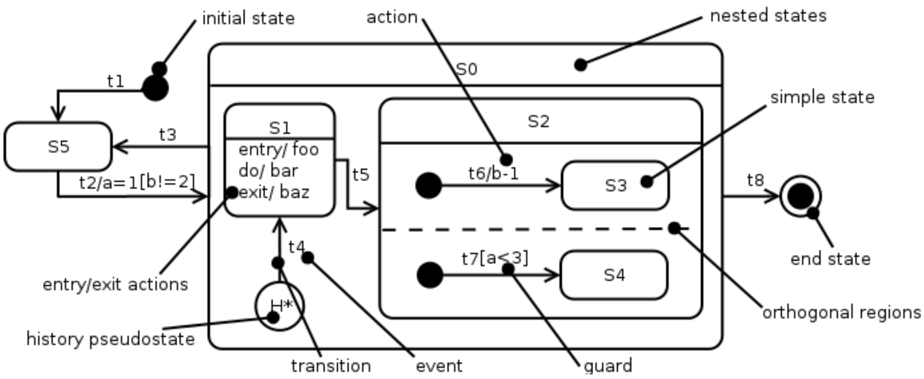

Figure 1.4. Example UML state machine

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 mean that if a system is in a substate then it is also at the same time in all the substate's superstates.

Orthogonal regions are regions, which are either in 'OR' or 'AND' 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 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.

Test selection criteria control the test case generation. MBT taxonomy includes the following identified criteria. *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* are guided 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.

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 [12]. Testing policies give rules for testing, while strategies are high-level guidelines.

Major tasks of test planning consist of

- 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 has to be fully formalised, so that a test generator is capable of generating test cases based on this formalisation and the software model.

1.3 Test generation

One of the most important thing that defines the test case generation is the chosen technology, because it has a strong impact on the effectiveness of software testing [2] [3]. That's why this topic is under active research and resulted different approaches.

Model-based testing taxonomy consists of the following popular test generation methods. 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 counterexample, 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 is guided by test case specification to reach a specific goal. *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.

We can see, that there are lot of possibility to choose from, when generating test cases for a given SUT. These methods all have advantages and disadvantages and we need to investigate them thoroughly to choose a suitable one for our needs.

1.3.1 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 already 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.

1.3.2 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* [25]. 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 "Eulerising" the graph.

The *New York Street Sweeper Problem* is a variant of the previous graph theory problem. It applies to directed graphs. 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 eulerise 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 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 are using a fitness function to find input parameters that maximises the achievement of test goals, while minimising testing costs.

1.3.3 Traditional MBT techniques

These test generation technologies include three similar solution especially for model-based testing purposes.

Model checking is a traditional MBT test case generation technique, where a model checker is used to generate test cases. Input of the model checker are the model of the SUT and the formalised versions of test criteria to check. During the procedure of proofing, if test criteria are valid in the model, witness traces and counterexamples are generated. A witness trace is a path, which consists of states, where the criterion is satisfied, while counterexamples represent a path, where the criterion is violated. The resulted paths can be used as set of test cases.

There are two main approaches in this topic, which are influenced by the chosen modelling notation (Section 1.1):

- **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.

Theorem proving is used traditionally to validate logical formulas. However model-based testing can also benefit from the power of this method.

Axiomatic foundations of MBT are based on some form of logic calculus. The models of the SUT is specified with logical expressions that are partitioned into equivalence classes. Each resulted class defines a specific feature of the SUT, therefore it represents a particular test case.

A possible partitioning can be, where the logic formula is transformed into disjunctive normal form (DNF) and solved with a higher-order logical theorem prover. Another way can be to transform the problem into solving finite state machines.

Constraint solving is used in a way, where a solver generates test cases by satisfying given constraints over a set of variables. With this method input model of the software and the test criteria are specified using constraints. The created constraints can be solved several ways for example with Boolean solvers (e.g. SAT solvers) or with numerical analysis (e.g. Gaussian elimination).

1.3.4 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* is 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 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 prove 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 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, which can not be solved with the today available tools. These formulas can be simplified by replacing solvable subformulas with concrete values.

1.3.5 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. Samples usually consist 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:

$$CA = \langle N, t, k, v \rangle$$

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.

1.4 Test execution

Test execution includes several steps, because the abstraction level of the generated test cases differs from the SUT. Therefore a previously mentioned adapter 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 following:

- Execute the complete test suite or individual test cases with test scripts.
- Log the outcome of the execution and report the identities, 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*).

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.

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 MBT has to help in the whole testing process. That means creating and verifying the model, generating test cases, constructing test scripts, adapters and oracles.

Utting, Pretschner and Legeard [30] 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 thesis.

Shafique and Labiche [27] 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 software 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. I tried to choose tools to research based on different categories. I wanted to start with an easy to understand software and continue with complex solutions. Besides that, I chose testing tools from different sources: open source, industrial and academic solutions as well. 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 parts 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 [20], 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 adapter has to be filled with the linking logic to 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 [18][19] 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 [13][14] 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 supports 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 seen on Figure 2.1.

1. First step is specifying requirements. Conformiq supports a huge amount of industrial requirements modelling tool (e.g. IBM Rational, Rhapsody, Sparx Systems Enterprise, ArchitectHP Quality Center, IBM RequisitePro, DOORS), but it also contains an own editor. 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 state charts and Java like action language.

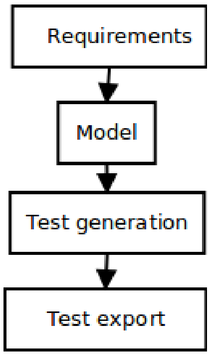

Figure 2.1. MBT process in Conformiq

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 [8] is a framework that consists 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, which maps to specific SUT methods.

2.5 ParTeG

Partition Test Generator [31][32] is an open source Eclipse plugin that can generate test cases from UML models annotated with OCL guards. It traverses 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 consequence. From our point of view the most important parts of their features are the used modelling notation and the test case (TC) generation methods, because these are the most crucial parts of the design. I summarised the collected information in the Table 2.1.

Name of the tool	Model	Intermediate model	TC generation method
GraphWalker	FSM	graph (GraphML)	SBST, combinatorial, random
PyModel	FSM + Python	graph	SBST, 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

- 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 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.

- The intermediate model is just always some kind of graph representation, 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 adapters, 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.

- 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, which 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 support 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 the conclusions of the investigated testing tools. This framework is need to be complete regarding the whole MBT process and has to help in all phases of the testing.

First the different testing phases will be discussed separately as at the model-based testing process specification (Section 1). Later I will describe the high level design of the framework and the used technologies.

3.1 Design choices

3.1.1 Modelling

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 is 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 lack 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. The EMF platform will be discussed in more details in Section 3.2.

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.

PLCspecif

PLCspecif [5] 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 (BME) 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.

The specification organised into modules (Figure 3.1), 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` is 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.

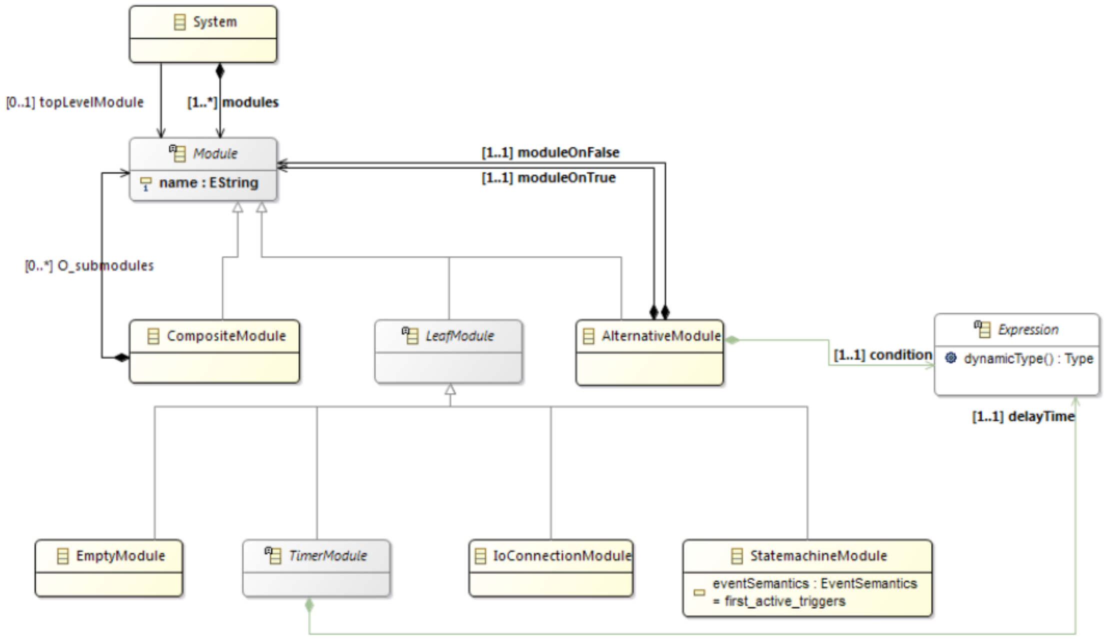

Figure 3.1. Module structure of PLCspecif [5]

From these module types, we are interested especially in the state machine notation. As shown on Figure 3.2 the metamodel is similar to UML state machine's metamodel, described previously (Subsection 1.1.2).

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.

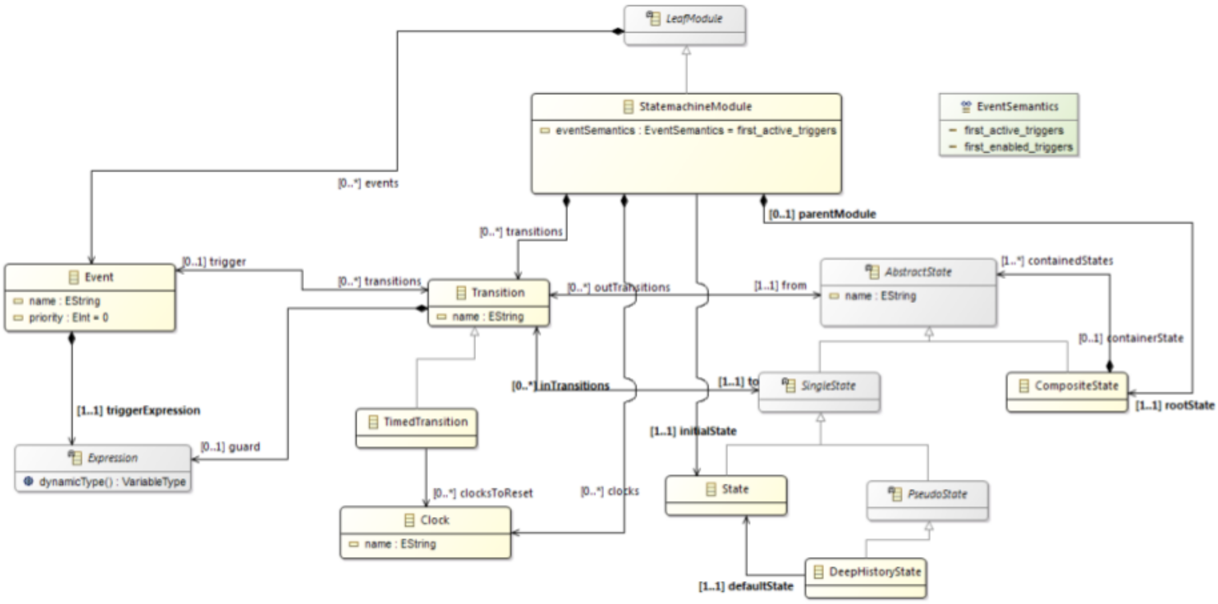

Figure 3.2. Structure of `StatemachineModule` [5]

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 3.3). 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.

Conclusion: 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 the syntax should conform with the original UML denotation.
- PLCspecif is rather a subset of the UML state machine language, 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.

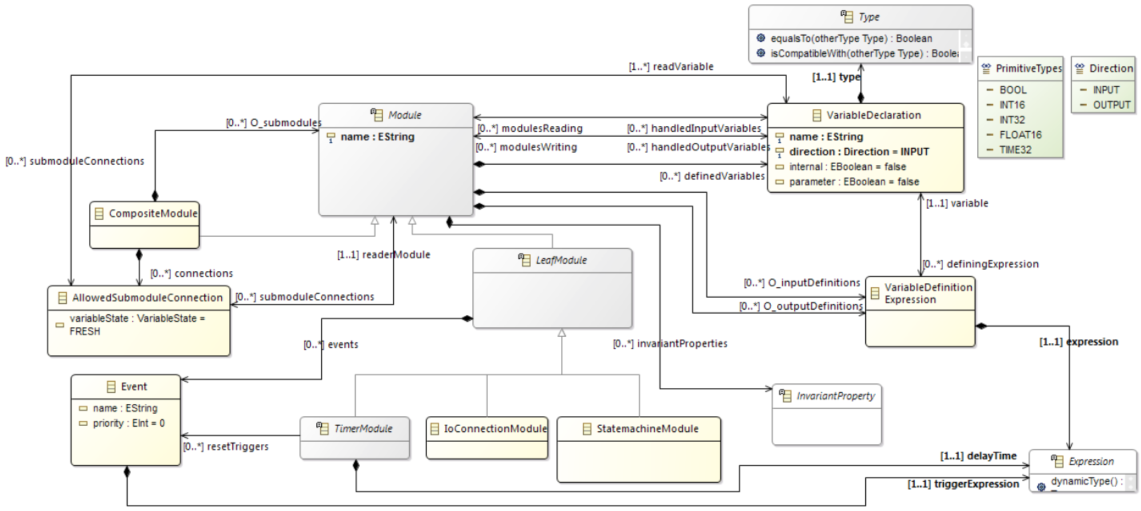

Figure 3.3. Variables in PLCspecif [5]

- 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.

3.1.2 Test planning

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

The scope of a generated test suite is always 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 related works (Chapter 2) 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 (Chapter 4).

3.1.3 Test generation

The third step is the test case generation. After examining the available tools, we saw that 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.

Alloy

Alloy [16][15][10] 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 to describe and model 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 's behaviour. UML semantics can be imitated with an Alloy model, but Alloy can do even more. OCL is 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.

Basic elements of the Alloy language will be presented with some code examples (Listing 3.1). Signatures (noted with `sig`) are the basic modelling elements of the language. From line 1 to 4 the relation of the signatures are defined. `A` represents an abstract signature, which can not present in the model without being also either a `B` or a `C`. However `B` and `C` are mandatory signatures enforced by using the `one` keyword. Signature `D` consists of some `A` and `E` is associated with one `D`.

Listing 3.1. Example Alloy code

```
1  abstract sig A {}
2  one sig B, C extends A {}
3  sig D { a: some A }
4  sig E { d: one D }
5
6  fact { no e1, e2: E | e1.d.a != e2.d.a }
7
8  fun count[e: E]: Int { #{aa: A | aa in e.d.a } }
9  pred two[e: E] { count[e] = 2 }
10
11 run { some e: E | two[e] } for 2
```

On line 6 there is a `fact` statement, which is an explicit constraint on the model. This constraint is used to ensure, there are not two signatures `E` having the same set of signatures `A`.

On line 7 there is a function statement, defined by `fun`, which is a parametrised expression that gets simply inlined at every invocation. Here function `count` is used to get the number of `A` signatures in a signature `E` . On line 8 there is a predicate, noted with `pred` that represents a formula, which can be evaluated to a boolean expression. Predicates can be executed for example with a `run` statement that instructs Alloy to search for a model that satisfies the given predicate. The presented statement checks, whether there are models, where signatures `E` have two different signatures `A`.

Conclusion: 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 is easy to integrate into a tool written in Java.

3.1.4 Test execution

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

3.2 Software design

Considering the previously described requirements the testing framework should have the following use cases (see on Figure 3.4):

UC.1: Model creation / editing

- **Description:** The user is able to create and edit a test model that represents the SUT's behaviour.
- **Priority:** High
- **Risk:** High
- **Scenarios:**
  - **Main:** A PLCspecif instance model is created.

UC.2: Model validation

- **Description:** The user is able to validate the test model, whether it is well-formed and satisfies some given constraints.
- **Priority:** Medium
- **Risk:** Medium
- **Scenarios:**
  - **Main:** Model validation is successful.
  - **Alternate:** Test model contains some error.

UC.3: Test suite generation

- **Description:** The user can generate a complete test suite.
- **Priority:** High
- **Risk:** High
- **Scenarios:**
  - **Main:** A suitable test suite is generated for full transition coverage.
  - **Alternate 1:** A suitable test suite is generated for full state coverage.

Alternate 2: Test suite generation was unsuccessful, because the coverage criterion is unsatisfiable.

UC.4: Test suite execution

Description: The user can execute the generated test suite and see the results.

Priority: Medium

Risk: Low

Scenarios:

Main: Test suite execution is successful.

Alternate: Test suite execution is failed, because the SUT may contain some error.

As the PLCspecif modelling framework already realised the first two use case, the testing framework can concentrate on the other two use cases.

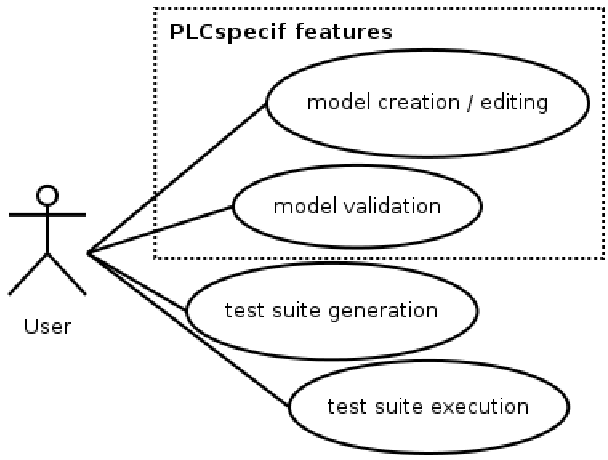

Figure 3.4. Use case diagram of the testing framework

Figure 3.7 shows the main components of the testing framework. PLCspecif is based on the EMF platform [7][29], but the testing tool can also benefit from the EMF toolset. The required two use cases can be realised with two Eclipse plugins. The first plugin will implement the the test suite generation and test execution tooling, the other will be the user interface of the testing tool. Before describing the behaviour and the implementation of the components in depths, the used EMF and the whole Eclipse platform have to be studied.

3.2.1 Eclipse Modelling Framework

Eclipse is an open source integrated development environment that contains a base workspace and a highly extensible plugin system [28]. The IDE was written mainly in Java and its primary use is for developing Java, C/C++, PHP applications but there are lot of other supported programming language and framework as well.

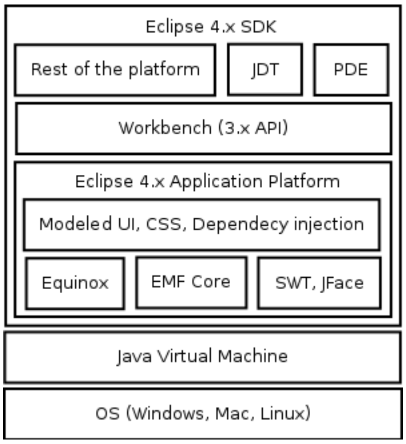

Figure 3.5. Architecture of the Eclipse platform [7]

The required tools for Java development are provided by the Java Development Tools (JDT), see on Figure 3.5. JDT includes Java editors, refactoring support, debugger, compiler and an incremental builder that recompiles only those files, which have changed or their dependencies.

Plug-in Development Environment (PDE) provides tooling to extend the capabilities of Eclipse through plugins. Eclipse gained its popularity and power at first from its plugin system. In fact everything is a plugin in Eclipse, except a small run time kernel. All feature are developed and integrated in the same way as a plugin, that's why third party developers are able to join and improve the Eclipse ecosystem.

Plugins are the base elements of the Eclipse component model. These plugins are in reality equal with OSGi bundles [1]. OSGi is a modular system and service platforms that implements a dynamic component model. The OSGi framework manages the bundles, their class loading and provides a dynamic, runtime lifecycle management.

OSGi supports runtime installation, starting, updating, stopping and uninstallation of bundles (Figure 3.6). When an application is started, bundles are in installed state. If all dependencies are met, then it changes to resolved state. Once a bundle is resolved, it can be started. Finally it becomes active and is able to interact with other bundles.

Essentially the OSGi bundles are JAR files with a manifest that describes the dependencies of the bundle. The only difference is between bundles and plugins that plugins have a `plugin.xml` file in their root directory, which contains metadata about the plugin. This

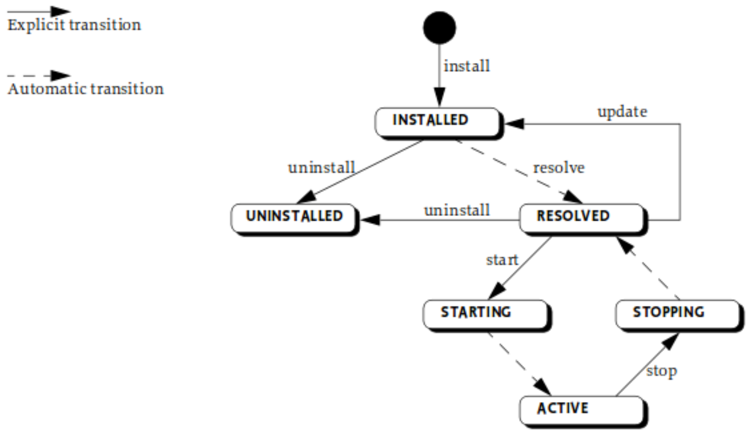

Figure 3.6. Lifecycle of OSGi bundles (Eclipse plugins) [1]

plugin manifest provides an other way to extend the features of a particular plugin, called extensions and extension points.

Extension points are considered public API that other developers can use to build their own extension. Figure 3.7 illustrates the components of the testing framework and which extension points they use to communicate with the Eclipse SDK. For example the test suite generator UI uses the `org.eclipse.ui.popupMenus` extension point to show a new menu item in the generated model editor's context menu.

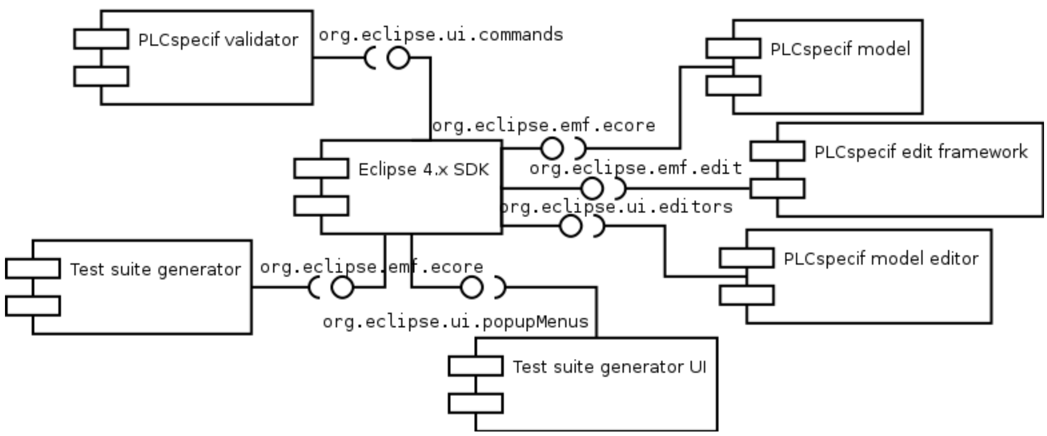

Figure 3.7. Component diagram of the testing framework

When Eclipse starts, the platform runtime (Equinox) scans the manifests of the available plugins and builds a plugin registry. These plugins are discovered at the startup, but only activated by the actual usage. This is called *lazy activation*. Lazy activation is enabled by the previously described OSGi platform, which results considerable performance improvements.

Eclipse user interface is built by two other important components of the platform. Eclipse Modeling Framework (EMF) is used to generate a model workbench that will be rendered into views. Default view renderer uses the Standard Widget Toolkit (SWT) to generate the code of the UI.

EMF is basically a modelling framework and code generation facility for building model-based tools. From a created model specification EMF is able to generate model classes, adapters for interacting with the model and a basic model editor. Models can be specified using annotated Java classes, UML or XMI. XMI (XML Metadata Interchange) is a standard to exchange metadata information and it integrates three OMG standard e.g. UML, XML and MOF (Meta Object Facility, for describing metamodels). EMF consists of three main parts: EMF (Core), EMF.Edit and EMF.Codegen.

EMF is based on a metamodel, called the Ecore metamodel, which can express other models by its components. Ecore is the key to take advantage of the entire EMF ecosystem. Thus Ecore usually used to define a custom metamodel specialised for the actual needs. The core package provides runtime support for models, including change notification, model persistence and an API to manipulate models.

EMF.Edit provides generic reusable classes for building editors for the previously created models.

EMF.Codegen can generate all part of a complete model editor. The first level of code generation is the model generation. This consists of Java interfaces, implementation classes and factories. On second level reside the adapter classes, which adapt the model classes for editing and display. The highest level is the editor, where a basic model editor is generated to start with.

EMF has tools for model transformation as well. There are frameworks that allow the engineers to use model-to-model and model-to-text transformations. For example ATL supports model-to-model, Acceleo [6] provides ways to transform models into text representations.

Chapter 4

Implementation

At the end of the design step it is clearly visible, what advantages the investigated technologies and tools have. The implementation of the created testing framework is represented regarding the phases of model-based testing process as earlier.

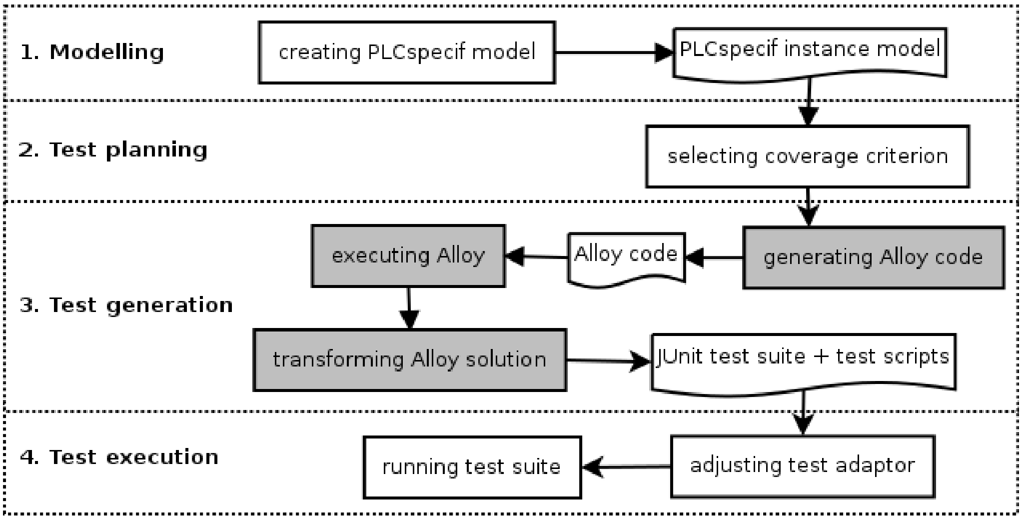

Figure 4.1. Usage of the test generator framework

Figure 4.1 demonstrates the usage of the testing framework. The manual operations are noted with white rectangles, the automatically executed operations are in grey rectangles. Outputs of the operations are represented with document symbols.

The different phases will be exemplified by the testing of a trivial application, which state machine is showed on Figure 4.2. This application has only one method that increases the value of a variable from 0 to 1. `T0` represent the single method the state machine has. Besides that, the state machine has an always satisfiable guard `G0`, as variable `output` is initialised to 0 and an event `E0` that increases the `output` variable by 1.

4.1 Modelling

First the input model of the testing framework has to be created. This is a PLCspecif instance model that is created with the default PLCspecif model editor, which is generated by the Eclipse Modeling Framework. PLCspecif's generated editor implements all the needed model creation, edition features (UC.1) and model validation as well (UC.2). Concerned components are noted on Figure 3.7, with PLCspecif model, edit framework, model editor and validator.

The abstract representation of the simplest possible state machine that has all the main UML state machine features is showed on Figure 4.3.

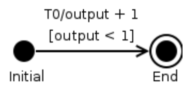

Figure 4.2. State machine of a trivial SUT

Constructed models should have some mandatory elements to fully support the test generation process e.g. the JUnit test case generation. Other elements are conventions that can be used to model a software's behaviour. The modelled system should have only one state machine module (`TrivialSM`), where the test model is realised. Each state machine should have a `VariableDeclaration`, named `output`, one `CompositeState` that contains all the `BasicStates`, from which two have the names `Initial` and `End` accordingly.

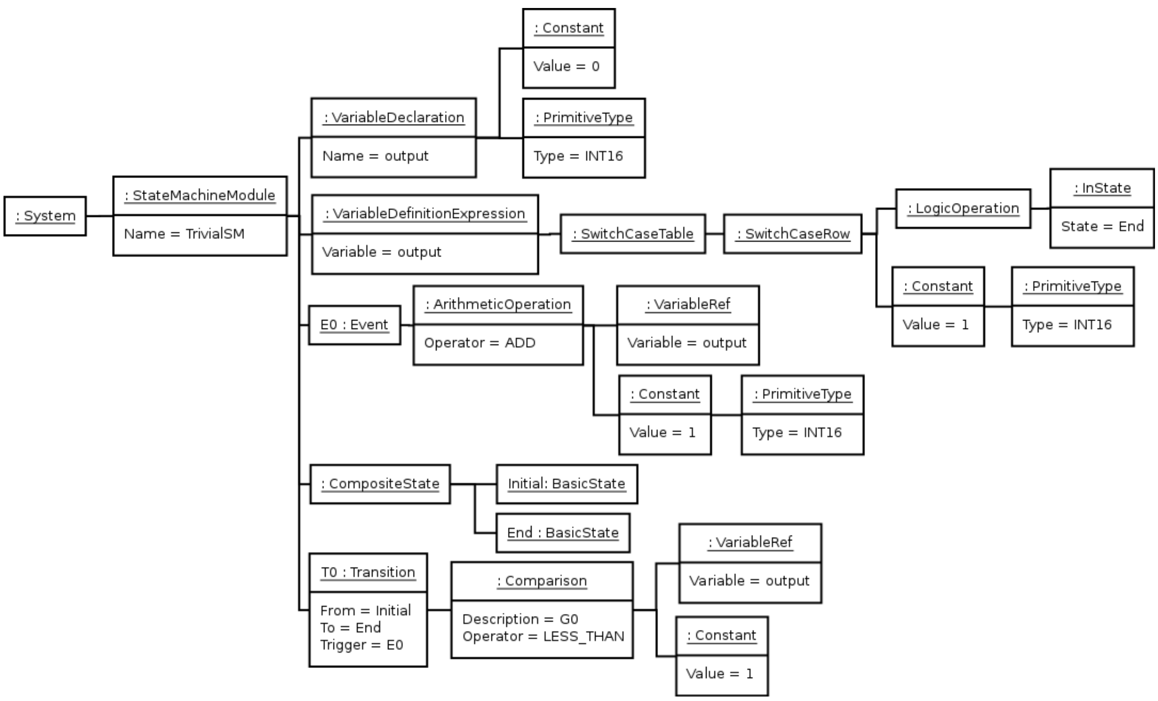

Figure 4.3. Object diagram of a PLCspecif instance model

Output assignments are described with a specific `VariableDefinitionExpression` for `output` variable. Outputs defined by a switch-case like structure that has as many `SwitchCaseRow` as states defined in the state machine. Each row has a condition definition (usually a `LogicOperation` with a check that verifies, whether the state machine is in a particular state e.g. `End` state) and a value definition (here represented by a simple `Constant`). In other words, the output of the state machine will be `1` in the state `End`.

Transitions can have guards (for example `G0`) defined on them using `Comparison` objects. Here `output` is checked, whether it is less than `1`. Transitions may have attached `Events`. For example `E0` increases the variable `output` by `1`.

4.2 Test planning

At this phase the user can select a test selection criterion (namely full state or transition coverage), which will be used by the test generation. This component is implemented in a separate Eclipse plugin (Test suite generator UI on Figure 3.7), as the selection is made on the user interface and it is always a good practice to decouple the UI from the business logic.

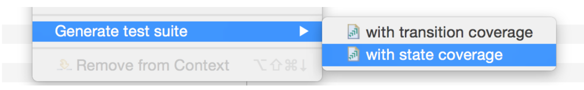

Figure 4.4. Screenshot of the test selection criterion

Eclipse SDK exports an extension point `org.eclipse.ui.popupMenus`, where the context menu of the application can be extended with new actions. So this Eclipse plugin contains two extensions, which use this interface to add two new popup menu items.

4.3 Test generation

When the user selects a test selection criterion on the user interface the test suite generation process will be started automatically. At first, the test generation problem will be transformed into Alloy code that can produce the test cases. The required information can be extracted from the previously created PLCspecif model and so the desired Alloy code can be generated automatically. This generation was solved with the model to text transforming capabilities of Acceleo and was implemented in a separate Eclipse plugin (Test suite generator on Figure 3.7).

The generated Alloy code has two main parts, the first one consists of statically generated code parts, the other includes highly dynamical, customised structures.

Listing 4.1 shows the static parts of the generated Alloy code. Basic structural metamodel of state machines is implemented with signatures (line number 1-3). `System` represents the internal state of the state machine by its state variables. `State`, `Transition` objects refer to the traditional EFSM elements. Each state knows its internal state through the `System` object and each transition connects two states.

From lines 5-14 the metamodel of basic testing objects are described. `Coverage` can refer to a state or transition coverage and so it can be transformed to a test suite. `Path` is set of method calls within the software. Generally more paths serve as a `Coverage`. `Step` is a method call in the context of software behaviour, but it also connects testing objects to state machine objects.

The helper function `steps` returns a relation containing all steps from a given path. From line number 16-28 are the basic rules of the system. Most of the rules describe the semantics of either the behaviour of state machines or the testing objects. Testing specific model consistency is also verified.

Listing 4.1. Static parts of the generated Alloy code

```
1  abstract sig System {}
2  abstract sig State {system: one System}
3  abstract sig Transition {from, to: one State}
4
5  sig Coverage { paths: some Path }
6  sig Path { firstStep: one Step }
7  sig Step {
8    from, to: one State,
9    via: one Transition,
10   nextStep: lone Step
11 } {
12   via.from = from
13   via.to = to
14 }
15 fun steps (p:Path): set Step { p.firstStep.*nextStep }
16 fact {
17   // test generation properties
18   all p:Path | one c:Coverage | p in c.paths // all paths belong to a coverage
19   all s:Step | one p:Path | s in p.firstStep.*nextStep // all steps belong to a path
20
21   // model consistency
22   all p:Path | p.firstStep.from = Initial // all paths start with an Initial state
23   all p:Path | one s:Step | s in steps[p] && s.to = End // all paths end with End state
24
25   // state machine properties
26   all curr:Step, next:curr.nextStep | next.from = curr.to // all steps are continuous
27   all sys:System | some s:State | sys = s.system // all systems belong to a state
28 }
29 pred inheritSystem(s1, s2: System) { s1 = s2 }
```

The predicate `inheritSystem` is similar to a helper function, but it can only return boolean values if its constraints are satisfiable. This predicate is used to pass along the internal state of the system between the different states.

Listing 4.2 shows the dynamic parts of the generated Alloy code. These parts of the Alloy code are generated using the previously created PLCspecif instance model (Figure 4.3).

Instance models of the `State` signatures (`Initial`, `End`) are instantiated using inheritance. Concrete transitions (`T0`) are inherited from the `Transition` object as well. Transitions that are connected to the initial state have to initialise the internal state variables of the state machine using the dynamically generated `initSystem` predicate. Guards (`G0`) and events (`E0`) connect to these transition objects too.

Listing 4.2. Dynamic parts of the generated Alloy code

```
one sig Initial, End extends State {}
some sig S extends System {
  output: Int
}
lone sig T0 extends Transition {}{
  from = Initial
  to = End
  initSystem[from.system]
  E0[from.system, to.system]
  G0[from.system]
}
pred E0(s1, s2: System) {
  s2.output = add[s1.output, 1]
}
pred G0(s: System) {
  s.output < 1
}
pred initSystem(s:System) {
  s.output = 0
}
```

Finally Listing 4.3 shows the test criteria formalisation by Alloy predicates. A possible test suite guarantees state coverage if all states are part of at least one step in the coverage. Transition coverage is defined in a similar way: a test suite guarantees transition coverage if all transitions can be mapped to a step in the coverage.

Listing 4.3. Formalising criteria with Alloy

```
pred state_coverage() {
  all s:State | some p:Path | s in steps[p].from + steps[p].to
}
pred transition_coverage() {
  all t:Transition | some p:Path | t in steps[p].via
}
run state_coverage for 10 but exactly 1 Coverage, 2 System
```

These criteria are applicable using Alloy's `run` statements. The constraint solving is executed in a bounded scope, that's why the scope of the search for examples need to be calculated dynamically.

The above described generated Alloy code is executed automatically using the Alloy Analyzer API. Operational parameters and other options were fine tuned to get the best possible performance from the integrated SAT solvers. The process of this research will be detailed in Chapter 5.

When the model is satisfiable and a possible coverage is generated, the solution is parsed into an internal model representation. This internal model is closely related to the testing level and is rather the metamodel of a JUnit test suite.

Similarly to PLCspecif, this internal notation uses a customised Ecore metamodel to describe its behaviour. Instead of starting with a default Ecore model editor, the metamodel is constructed dynamically using annotated Java interfaces. Based on them, an Ecore model and a generator model is constructed, which will generate the remaining code.

Listing 4.4. Generated JUnit test suite

```
package trivialsm;

import org.junit.Before;
import org.junit.Test;
import junit.framework.TestCase;

public class TrivialSMTest extends TestCase {
        protected TrivialSM trivialsm = null;
        protected TrivialSMTestAdapter adapter = null;

        @Before
        public void setUp() {
                trivialsm = new TrivialSM();
                adapter = new TrivialSMTestAdapter(trivialsm);
        }

        @Test
        public void testPath1() {
                assertEquals(1, adapter.T0());
        }
}
```

The Alloy solution parser is based on the *Builder* pattern [9] and uses model *Factory* classes from EMF.Edit to create the internal model. After building the test suite model, Acceleo transforms automatically this model to text representation including a complete JUnit test suite (see on Listing 4.4) with pure POJO helper classes.

Listing 4.5 shows a more complicated example. The generated test suite guarantees full state coverage for a software that represents a simplified stopwatch (Figure 5.2). In this example the test suite consists of two different test paths that are needed to guarantee the selected test criterion.

Listing 4.5. Test suite for a stopwatch (Figure 5.2)

```
public class StopWatchTest extends TestCase {
        protected StopWatch stopwatch = null;
        protected StopWatchTestAdapter adapter = null;

        @Before
        public void setUp() {
                stopwatch = new StopWatch();
                adapter = new StopWatchTestAdapter(stopwatch);
        }

        @Test
        public void testPath1() {
                assertEquals(1, adapter.on());
                assertEquals(2, adapter.start());
                assertEquals(3, adapter.stop());
                assertEquals(5, adapter.off());
        }

        @Test
        public void testPath2() {
                assertEquals(1, adapter.on());
                assertEquals(2, adapter.start());
                assertEquals(4, adapter.split());
                assertEquals(3, adapter.stop2());
                assertEquals(5, adapter.off());
        }
}
```

4.4 Test execution

Output of the test generation process consists of three scaffolded helper classes (showed on Figure 4.5). They are all generated automatically, derived from the generated test cases and the test model.

Test suite is generated as a standard JUnit test fixture (see `TrivialSMTest` in Listing 4.4). The different test paths are separated into annotated test methods (e.g. `testPath1`). An additional `setUp` method can be used to initialise the test adapter. Test oracles are generated dynamically in the form of single assertions and inserted into the test methods.

The generated test suite can contain more test paths, which include usually more test cases. A single path starts from the initial state of the SUT and ends in the end state. Set of the paths offer full state or transition coverage within the SUT.

When the user re-generates a test suite for the SUT, it can differ from the previously generated test suite, although the generated test suite will have the same structural properties as earlier. There is this anomaly, because the execution of the included SAT solver is undetermined.

Test adapter is scaffolded as a simple POJO class (`TrivialSMTestAdapter`). This class maps the model transitions to concrete method calls. Initialisation of the SUT can be done in the adapter's constructor as well. Method calls are wired automatically to concrete SUT's methods using a simple heuristic. If the mapping is not trivial, the adapter's code may not be perfect and needs some adjustment. Here the SUT, named `TrivialSM`, is quite simple and the adapter does not need any further modification.

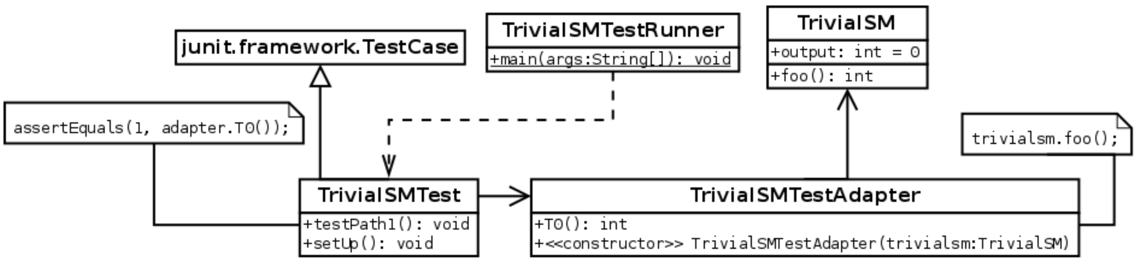

Figure 4.5. Class diagram of the scaffolded test helpers

Test script is a simple JUnit test runner (`TrivialSMTestRunner`) that reports the result of the testing. It works like the default JUnit test runner, so it reports out the result of the complete test suite execution (Listing 4.6).

Listing 4.6. Successful test suite execution output

```
Finished in 0.005 seconds
1 examples, 0 failures, 0 ignored
```

The test reporter tells the difference between the expected and the actual test outcome, moreover if some discrepancy is found, it shows the stack trace of the actual exception (Listing 4.7).

Listing 4.7. Failed test suite execution output

```
Finished in 0.007 seconds
1 examples, 1 failures, 0 ignored
Failed examples:
testPath1(trivialsm.TrivialSMTest)
junit.framework.AssertionFailedError: expected:<1> but was:<2>
...
```

4.5 Overview of the implementation

Project*	Package	Features
`core`	–	JUnit test suite generation.
`core`	`alloy`	Alloy generation and solution parsing.
`core`	`common`	Eclipse plugin specific classes.
`core`	`testing`	Test adapter model interfaces.
`core`	`testing.impl`	Test adapter model implementations.
`core`	`testing.util`	Test adapter model utility classes.
`ui`	–	UI wiring to business logic.
`ui`	`common`	Eclipse plugin specific classes.
`ui`	`popupMenus`	Popup menu actions on UI.
`measurements`	–	Utility classes for measurements.
`examples`	`scalabilitytest`	Scalability testing models.
`examples`	`statecoverage`	State coverage testing model.
`examples`	`trainsitioncoverage`	Transition coverage testing model.
`examples`	`stopwatch`	Simplified stopwatch model.
`examples`	`thread`	Java Thread class model.
`examples`	`trivialsm`	Trivial state machine model.
`examples`	`useraccount`	User account model.

*Each project lives in the namespace `hu.bme.mit.plcspec.testsuitegenerator`

Table 4.1. Created projects and packages

The created framework includes four Java projects. The `core` project contains business logic, while the `ui` project implements the user interface. These two projects can be installed in the same way as other Eclipse plugins. Moreover the framework contains some utility tool to measure the performance of the test generation and it also includes some usage examples.

During the development I used the following technologies and tools:

- PLCspecif (1.0)
- Alloy (4.2)
- Eclipse Modeling Tools (4.5.0)
- Acceleo (2.0)

Chapter 5

Measurements and scaling

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.

Execution of each testing configuration was measured 10 times and the results in the following section always refer to average metrics. The tests has been running on the configuration that can be seen 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. Alloy Analyzer supports huge variety of SAT solvers and they have a decent number of tuning possibility.

I was able to integrate seven different SAT solvers into the system, namely CryptoMiniSat [4], Glucose [11], MiniSat [23], MiniSat with core extraction, Sat4j [26], Lingeling [21] and its parallel version Plingeling. The another dimension of the measurement was the SAT solver configuration. Settings of two configurations were not obvious, that's why I chose to measure these parameters.

The first investigated option was Skolem-depth that controls the maximum depth of alternating universal and existential quantifier, when generating a Skolem function. Minimum value is 0, which means it will only generate Skolem constants and will not generate Skolem functions. Maximum value is 4, where Skolem functions are generated in depth of 4.

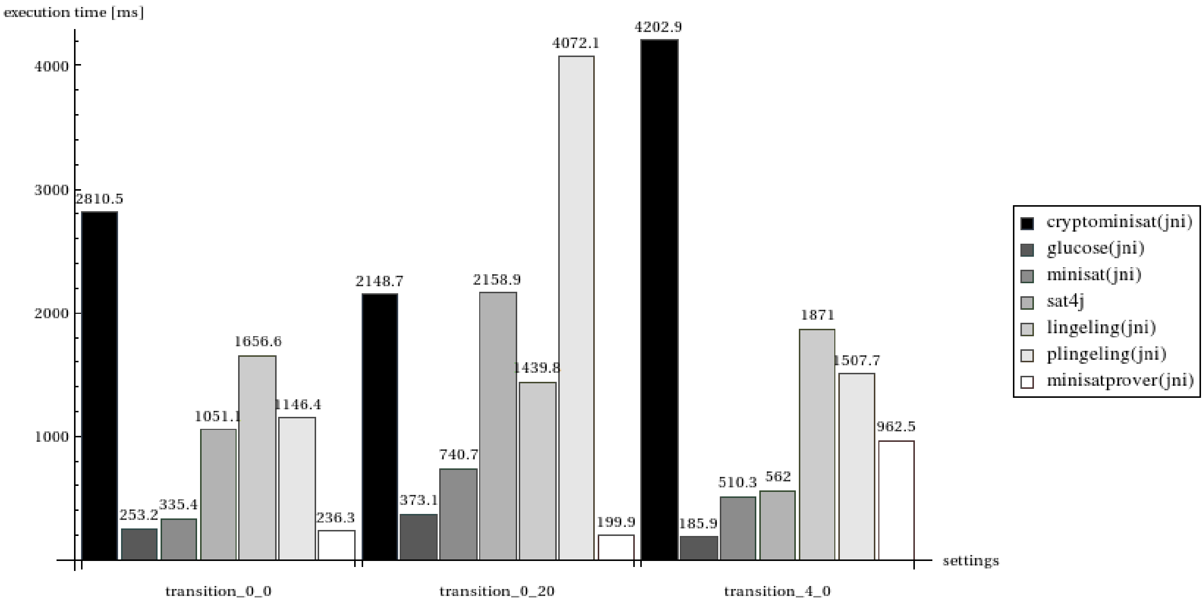

Figure 5.1. Adjusting Alloy settings

Second option to inspect was symmetry breaking. The official documentation suggests that if a formula is unsatisfiable, then in general the higher this value, the faster the solver will finish. On the other hand, if the formula is satisfiable, then the value should be set to a lower value. Minimum of symmetry breaking is 0, maximum is 20.

Figure 5.1 shows the result of the measurement. On the x-axis are the solver configurations in the form: *<test selection criterion>_<Skolem-depth>_<symmetry breaking>*. On y-axis is the execution time and the different colours representing different SAT solver implementations.

The result was similar with state and transition coverage criteria as well, therefore here only the transition coverage version is demonstrated. Measurement with Skolem-depth of 4 and symmetry breaking of 20 is not presented, because this configuration could not satisfy the given problem in acceptable period of time.

The fastest solver became the award winning Glucose SAT solver. Only the MiniSat solvers could approach the performance of Glucose, the other solvers seemed to be significantly slower. That's why Glucose became the default solver of the testing framework.

Symmetry breaking with a value of 0 was also a clear choice, since the documentation suggests that the programs solves satisfiable problem with a lower value more easily.

Changing the Skolem-depth did not impact the speed very much, only a small amount of difference could be measured. Glucose was faster with an option 0, which was also the default value, so I did not change that.

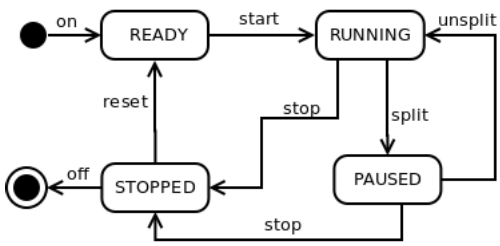

Figure 5.2. Stopwatch FSM for testing

Later, as I improved the test generation algorithms, I checked, whether these options still held. The framework was still the fastest with this configuration, which was also not surprising, because Alloy documentation recommends this configuration regardless of the model complexity.

As an input for the first two measurements I used an example FSM that represents a simplified stopwatch behaviour (Figure 5.2).

5.2 Optimisations

After the first measurements, the performance of the test generation seemed to be adequate to work with, but as I increased the complexity of the problems to solve, so increased the execution time of the test generation.

The input model, as I mentioned earlier was the stopwatch FSM (Figure 5.2) that 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.

In the first development iteration, the test generation algorithm could not generate test cases for this FSM within a tolerable time. This was a huge problem, as it could jeopardise the success of using constraint solving methods for test generation. After examining the test generation process and the results with Alloy Analyzer, I identified the main issues.

The first problem was that the model was overconstrained and that's why the SAT solver generated to much variables, while parsing the given constraints. The other problem was that the solver generated different objects in a scenario for the same purpose, that's why state space increased exponentially.

Solution for these problems is the same: the Alloy model needs to be simplified. This has been done in two steps, Figure 5.3 shows the effects of the optimisations. After deleting the unneeded constraints from the model, the Glucose SAT solver could solve the stopwatch problem. The execution time at this time displayed with the label "Optimised version 1" on Figure 5.3. Later I modified the model to reuse the previously generated model objects to reduce the state explosion. Final version is showed with the label "Optimised version 2".

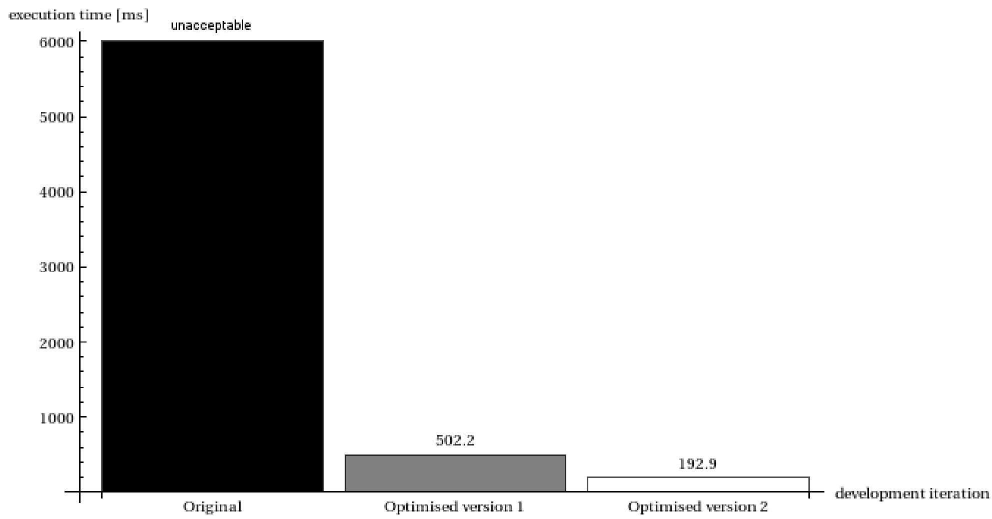

Figure 5.3. Optimisations results

The speed of the execution increased radically during these optimisations. In the first step, the increase can not be calculated, as the speed was unacceptable in the first version. Considering the speed of the chosen SAT solver, the execution became twice as faster, than previously.

5.3 Scalability

In the first measurement I investigated the scalability of the created testing framework. I assumed that the test generation algorithm does not scale equally regarding the number of states or transitions generated during the test case generation. That's why I measured the performance considering these two options.

First, applicable PLCspecif models have to be generated in different sizes that are input models of the framework. Then the test generation process can be started using these models, where the execution speed is measurable.

Creating a random PLCspecif model that can represent a SUT, where different test selection criteria can be satisfied is not an easy task, as it involves complex graph theory algorithms to generate such a graph representation. Main parts of the process are the following:

1. At first a graph model of the SUT has to be generated. Process of the graph generation is demonstrated in Algorithm 1.

Algorithm 1: Generating SUT models based on the size of states or transitions

---

Data: n, t such that t is the type of the generation scope for a $G(V, E)$ graph,
and $n = \begin{cases} |V|, \text{ if } t = \text{state} \\ |E|, \text{ if } t = \text{transition} \end{cases}$

Result: $G(V, E)$ graph representation of the SUT

1 $s \leftarrow 0$;
2 **while** $s \neq n$ **do**
3     $in\_degree\_sequence \leftarrow [0] + \text{random sequence} + [1]$;
4     $out\_degree\_sequence \leftarrow [1] + \text{random sequence} + [0]$;
5   **while** $|in| \neq |out|$ **do**
6         $out\_degree\_sequence \leftarrow [1] + \text{random sequence} + [0]$;
7   **end**
8     $G(V, E) \leftarrow$ random directed pseudograph, using the degree sequences;
9   **if** $\exists e \in E : e = (v, v) : v \in V$ **then**
10         $E \leftarrow E \setminus e$;
11   **end**
12     $s = \begin{cases} |V|, \text{ if } t = \text{state} \\ |E|, \text{ if } t = \text{transition} \end{cases}$
13 **end**

The graph will be generated randomly using previously defined in/out-degree sequences. FSM models always have an initial and an end state, which have an in-degree 0 and 1, and out-degree 1 and 0 consequently. The generated model has to support one of our test selection criteria to be able to execute later with our testing tool, so I chose to implement state coverage support on the generated models, as it is more simple, than the transition coverage. FSMs having a full state coverage need to have all state with incoming and outgoing transitions, thus all state will be reachable. These rules are formalised in lines 3-7.

Generated graph should not have self-loops, as triggering transitions always initiates a state change (line number 9-11).

The resulted graphs was exported into GraphML format.

2. Exported graphs are transformed into PLCspecif model notation using generated factories offered by the EMF platform. Skeleton model of a SUT is created by code, which is filled later with the states and transitions coming from the parsed GraphML graphs.

Figure 5.4 shows the results of the scalability measurements. As I assumed previously the framework scales differently regarding the number of states or transitions that the SUT has.

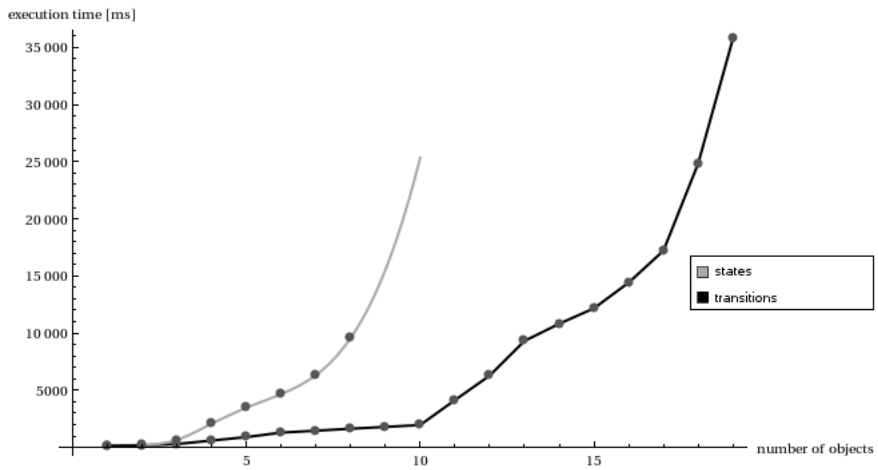

Figure 5.4. Scalability results regarding the number of states or transitions

The created model-based testing framework performs well, while generating test suites for medium sized models. Practically that means SUT with 5-10 states and 15-20 transitions are solvable by this tool. In this range the execution speed varies under 35 seconds, which is an acceptable speed.

Used input models were randomly generated test models, that's why the complexity of solving these problems does not perfectly follows the increase of the concerned testing objects. Besides that, we can see a trend line, that the execution speed increases linear or polynomial at first, but later it will grow exponentially regarding the number of states or transitions.

Important to note that the number of states in the SUT model, affects the execution speed more, than the number of transitions, as possible states of the `System` (set of internal variables) are bound to the `State` objects in the Alloy model. So as the number of states increases, so does the possible value assignments for internal variables increases.

Chapter 6

Summary and further development

In this chapter first I position and categorise the created model-based testing software in a way presented in Chapter 1 and in Chapter 2. Utting, Pretschner and Legeard [30] defined a taxonomy for categorising tools based on their test model and the used test generation algorithms, while Shafique and Labiche [27] identified the key features of MBT tools and classified them according to these criteria. Showing what the finished tool is capable of can be presented easier using these classifications.

Summarising the work is followed by the possibilities of further development and improvements. These statements are either proven with the results of the measurements or defined by the need for new features that the available tools lack of.

Finally I will end this thesis with the final results and personal experiences.

6.1 Positioning of the thesis

The taxonomy and the tools review protocol describe the most important aspects of a model-based testing tool. We can use the same methods to evaluate the developed testing tool and see how it differs from other tools.

Table 6.1 shows the results of this evaluation. On one hand the created tool is very similar to the available testing tools. Most of them use some FSM or UML like modelling notation. This is not so surprising, because one of the main goal of this thesis was to generate test suites for state-based models, which implicated to use untimed, deterministic, discrete transition based models. Preferring structural model coverage criteria, moreover concentrating on model-flow criteria is a similarity as well. Nevertheless implementing such a criteria is the easiest, that's why support most of the available testing tools them, when using state based models.

Property	Value	Notes
Subject of testing	*SUT*	The test model represent the SUT, not its environment.
Test model separation	*Separated*	Different model is used for testing and development.
Model characteristics	*Deterministic*, *untimed*, *discrete*	Possibility to support timed transitions.
Model paradigm	*Transition based*	PLCspecif state machine notation.
Test selection criteria	*Structural model coverage*	Full state and transition coverage are supported.
Test generation technology	*Constraint solving*	Generating test cases using Alloy.
Test execution	*Offline*	-
Model-flow criteria	*Transition* and *state coverage* are supported; transition-pair, all-path coverage are not. Sneak-path and scenario coverage are not applicable, because the model is always considered complete and modifying the search for test cases is impossible, because the test generation technology.	
Script-flow criteria	*Interface*, *statement*, *decision* coverage are supported implicitly, others are not applicable, because the test generation technology.	
Data and requirements criteria	Not supported.	
Test scaffolding criteria	*Adapter creation*, *oracle automation* are supported. *Stub creation are not supported.*	
Related activities criteria	*Model creation*, *model verification*, *test case debugging*, *test case debugging*, *regression testing* are fully supported. Requirements traceability is not supported.	

Table 6.1. Applied techniques and supported features of the created tool

On the other hand this new testing tool is unique. Constraint solving as test generation technology is a rare choice. Though it is a useful method by test generation as we saw earlier. Other strengths of this tool are the full support of test scaffolding and related activities criteria. From the reviewed tools only a few can support all these features in one integrated framework.

6.2 Possibilities for further development

Better scalability After some development iteration the created framework is able to solve real world problems, but as the number of state increases in the SUT model, the tool's execution speed enlarges exponentially.

On the one hand it can be problem if the goal is to generate test suite for a complex SUT, but on the other using complex state machine features in a big SUT model is certainly not recommended. When developers use a state machine like model, usually the used state machine are not so complicated.

Either way the execution speed can be improved. The improvement can be achieved with more option:

- The most time consuming part of the system is the Alloy test generation, which can be improved by generating less runtime objects, while solving a SAT formula. Alloy model has to be simplified to do this.
- Input model can be simplified before transforming to Alloy too. Solving easier parts of the models separately can also reduce the state space, therefore the constraint solving part will be faster. The easy parts of the system can be solved with other test generation methods e.g. with graph algorithms.

Support more PLCspecif features Currently the testing tool supports most of the elements from the PLCspecif feature set. However some basic UML like element are not supported, for example composite states and pseudo states as deep history states.

Guards and events also can not be defined arbitrary. These elements may benefit from the usage of the whole expression model. Expressions have an extensive metamodel and currently only the binary operations are supported. Supporting the other operations may improve the usability of the testing framework.

Requirements traceability Requirements traceability can be defined as documenting the life of a requirement. In the field of software testing it also means the reporting of the requirements coverage.

Possible scenarios to support this feature can be the following:

- Creating a traceability matrix may help to check if the current requirements are being met or can help in the creation of a requirements specification. When a requirements is accomplished it should be noted on test models. Regarding these information traceability matrices can be filled during the test case generation.
- Third party requirement management tool integration can also help in a similar way as traceability matrices.

6.3 Conclusions

Regarding the previously defined requirements, I successfully created a model-based testing framework based on state machine models.

- I presented the main goals of model-based testing and the general testing process.
- I investigated the related work that use state machine models for generating tests.
- I chose a state machine modelling notation and designed a framework that is able to generate complete test suites with previously defined test selection criteria.
- I implemented the testing framework and described its internal behaviour.
- Finally I evaluated the finished solution and sketched some possible future work.

After the testing and measurements phases I have concluded that the resulted testing framework is able to complete the tasks and to satisfy the requirements that were defined at the start of the work. The software can maybe fill a gap, because it was designed to offer solutions to problems, that the other tools can not solve.

During the development of this software I continuously learned new technologies and algorithms. Finally I can say that it was a wonderful experience to design and develop a complete and useful solution for such a complex and hard problem.

List of Figures

List of Tables

List of Code Listings

List of Algorithms

Bibliography

[1] OSGI Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. Technical report, OSGI Alliance, 2009.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *The Journal of Systems and Software*, 86:1978–2001, 2013.

[3] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. Part iii. model-based test case generation. In *Model-Based Testing of Reactive Systems*, pages 277–279. Springer Berlin Heidelberg, 2005.

[4] CryptoMiniSat. Máté Soós. `http://www.msoos.org/2014/04/cryptominisat-4-released/`, 2015. [Online; last access 2015.11.15].

[5] István Majzik Dániel Darvas, Enrique Blanco Viñuela. Syntax and Semantics of PLCspecif. Technical report, European Organization for Nuclear Research, 2015.

[6] The Eclipse Foundation. Acceleo. `https://eclipse.org/acceleo/`, 2015. [Online; last access 2015.11.15].

[7] The Eclipse Foundation. Eclipse Modeling Framework. `https://eclipse.org/modeling/emf/`, 2015. [Online; last access 2015.11.15].

[8] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. *ACM SIGSOFT Software Engineering Notes*, 27:134–143, 2002.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] Ana Garis, Ana C.R. Paiva, Alcino Cunha, and Daniel Riesco. Specifying uml protocol state machines in alloy. In *Integrated Formal Methods*, pages 312–326. Springer Berlin Heidelberg, 2012.

[11] Glucose. Gilles Audemard, Laurent Simon. http://www.labri.fr/perso/lsimon/glucose/, 2015. [Online; last access 2015.11.15].

[12] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008.

[13] Conformiq Inc. *Conformiq User Manual*. Conformiq Inc., 4.4 edition, 2011.

[14] Conformiq Inc. Conformiq. https://www.conformiq.com/, 2015. [Online; last access 2015.11.15].

[15] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, 2002.

[16] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[17] Daniel Jackson. Alloy. http://alloy.mit.edu/alloy/, 2012. [Online; last access 2015.11.15].

[18] Jonathan Jacky. Pymodel: Model-based testing in python. In *Proceedings of the 9th Python in Science Conference (SciPy 2010)*, pages 1–6, 2010.

[19] Jonathan Jacky. PyModel. http://staff.washington.edu/jon/pymodel/www/, 2013. [Online; last access 2015.11.15].

[20] Kristian Karl. GraphWalker. http://graphwalker.org/, 2015. [Online; last access 2015.11.15].

[21] Lingeling. Armin Biere. http://fmv.jku.at/lingeling/, 2015. [Online; last access 2015.11.15].

[22] Zoltán Micskei. Modell alapú automatikus tesztgenerálás. Master's thesis, Budapest University of Technology and Economics, 2005.

[23] MiniSat. Niklas Eén, Niklas Sörensson. http://minisat.se/, 2010. [Online; last access 2015.11.15].

[24] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. Technical report, Object Management Group, 2011.

[25] Harry Robinson. Graph theory techniques in model-based testing. In *Semantic Platforms Test Group, Microsoft Corporation, Presented at the 1999 International Conference on Testing Computer Software*, pages 1–10, 1999.

[26] Sat4j. Daniel Le Berre, Anne Parrain. `http://www.sat4j.org/`, 2013. [Online; last access 2015.11.15].

[27] Muhammad Shafique and Yvan Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2013.

[28] Jeffrey M. Squyres. *The Architecture Of Open Source Applications*. Self published, 2012.

[29] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[30] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012.

[31] Stephan Weißleder. Test models and coverage criteria for automatic model-based test generation with uml state machines. Master's thesis, Humboldt-Universität, 2010.

[32] Stephan Weißleder. ParTeG. `http://parteg.sourceforge.net/`, 2014. [Online; last access 2015.11.15].

[33] Karolina Zurowska. Language specific analysis of state machine models of reactive systems. Master's thesis, Queen's University, 2014.