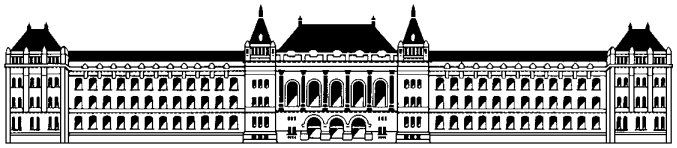

Budapest University of Technology and Economics
Department of Measurement and Information Systems

TEST GENERATION BASED ON STATE MACHINE MODELS

Project Laboratory 1 final report
2013/14. I. semester

MILÁN GYÖRGY UNICSOVICS (M9GNTV)

I. year, computer engineering student
MSc Specialization in Dependable System Design

Consultant:
Dr. Zoltán Micskei assistant professor, MIT

Contents

1 Introduction

The main goal of software testing is fault detection, where we compare the software's intended and actual behaviour to make sure there are not any difference between those, regarding the requirements.

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

Model-based testing substitutes the traditional ad-hoc software testing methods which relies on behaviour models that describe the intended behaviour of the system and its environment. From the models set of test cases are generated automatically and then executed on the tested software.

My research aims to prepare to create a new automated testing framework for software based on state machine models. Before that related works and similar solutions have to be examined. The result of the research later can be used to design and develop a software which fills the need of a fully automated model based testing framework.

2 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 [8]. This section introduces the concept of this variant of software testing through a concrete process (Figure 1).

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

2. Test selection criteria decide how the test cases are chosen, which point of view is important by testing. Further details are at the subsection 2.1.

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

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

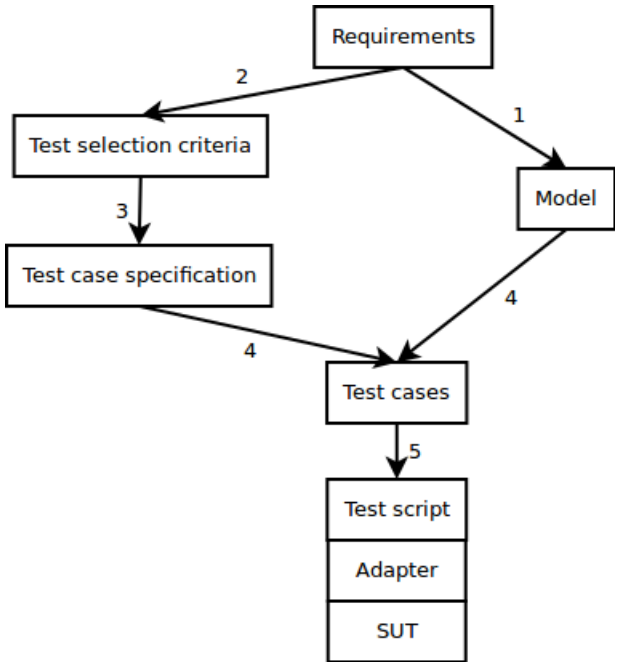

Figure 1: Model-based testing process

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

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

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

2.1 Taxonomy

Utting, Pretschner and Legeard investigated the currently available MBT solutions and defined (Figure 2) a taxonomy which concentrates to three major properties of model-based testing. The three dimensions of their taxonomy are the modelling specification, test generation and test execution.

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

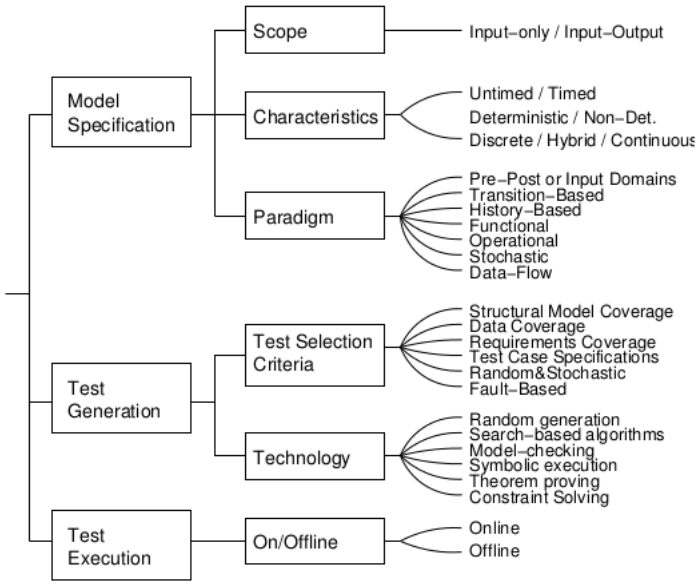

Figure 2: Model-based testing taxonomy [8]

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

Model paradigm The third dimension is the paradigm that is used to describe the model. *State-based notation* means, that set of variables defines the model, which represents the internal state of the system. By *transition-based notation* the model focuses on the transition between the state of the system. Finite state machines are examples for this paradigm. *History-based notations* model the allowable traces of its behaviour over time. By *functional notation* collection of mathematical functions model the system. *Operational notations* describe the model as a set of executable processes running parallel. *Stochastic notations* describe the model by a probabilistic model, as it is rather suitable to model the environment than the SUT itself. The last paradigm is the *data-flow notation*, where the main concept is the concentration to the data, rather than the control flow.

Test selection criteria Test selection criteria control the test case generation. *Structural model coverage criteria* aim to cover a part of the model, for example nodes and arcs of the transition-based model. The basic idea of *data coverage criteria* is to split the data space to equivalence classes and choose values from them. *Requirements based coverage criteria* are linked to the informal requirements of the SUT and it applies the coverage to the requirements. *Ad-hoc test case specifications* guides by the test case specifications. *Random and Stochastic criteria*

are useful rather to model the environment and applicable to use with a stochastic model. *Fault-based criteria* can be very efficient, because it concentrates to error finding in the SUT.

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

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

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

3 Tools review

Model-based testing is a mature idea, and it has an extensive literature. Nevertheless the number of the available tools is less than we can expect that. To really take advantage of model-based testing, reliable tools and automation support are required. A usable model-based testing tool has to help in the whole testing process. That means creating and verifying the model, generating test cases, constructing test scripts, adaptors and oracles. Utting, Pretschner and Legeard [8] defined MBT as testing that relies on models specifying the intended behaviour of the SUT. In reality that would mean restricting MBT to black-box testing, where we can only generate abstract test cases from the behaviour model. That's why Shafique and Labiche defined MBT as a support of software testing activities from a model of the SUT behaviour. We follow this point of view in this paper.

Shafique and Labiche [7] collected the available tools that rely on state-based models and created a systematic review considering the previously and newly defined criteria.

Model-flow criteria This criterion details the state-based coverage options and applicable to state-based models, which belong to transition-based models. The coverage options can be state, transition, transition-pair, sneak path, all-paths and scenario criteria. The first five are well-known, scenario criteria means, that the test should follow user defined test sequence to pass.

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

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

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

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

3.1 GraphWalker

The first investigated tool was GraphWalker [4], which can create online and offline tests from finite state machines, extended finite state machines or from both of them. The framework is written in Java, the related tools belong to Java world as well. Maven is used to run the tests, TestNG to describe the test cases.

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

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

3.2 PyModel

PyModel [2][3] is an open-source MBT testing framework written in Python. It consists three main tools:

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

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

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

PyModel's test input has to be created by code. The methods will be the transitions in the FSM, states are the defined attributes. It is possible to combine different models in a test. Scenarios supported as well, so user can guide the tests with a given test case sequence. There are two test coverage criteria, state-based and transition-based coverage.

3.3 Conclusion

After examining the available tools the result is the following:

- The tools implement different coverage criteria, but mostly just the easiest ones (state-based and transition based version of structural model coverage criteria). More difficult criteria are avoided, for example data coverage, requirement-based and fault-based criteria and transition pairs coverage from structural coverage criteria.
- Script-flow criteria are rarely used techniques. Only a few tools support guards on transitions or use scripts for example to give control information of the SUT.
- Scaffolding solutions of the tools are incomplete. Fully automatic generation of test adaptors, oracles are seldom supported.
- Regression tests are not supported.

4 Connection with graph theory

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* [6]. Given that it is impossible to cross each edge once in an undirected graph during a graph walk, in other words it does not have an Eulerian tour. What is the minimal amount of re-crossing we need to create a walk that uses each edge? The solution is to duplicate the shortest edges between the vertices having odd degree. This process is called "Eulerizing" the graph. In model-based testing one can use this idea, by creating a transition-based model, which can be represented as a graph. The vertices are the states of the SUT and the edges are the callable methods. A generated Eulerian tour gives a full transition-based structural model coverage.

The *New York Street Sweeper Problem* is a variant of the previous graph theory problem. It applies to directed graphs, and arcs need to duplicate to reach, that each nodes have out-degree minus in-degree equal zero.

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

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

5 Implementation of an MBT framework

After studying the theory of model-based testing, I started to plan a framework that takes advantage of the learned algorithms and realises them in practice. The framework is written in Python language and uses FSM models in GraphML format.

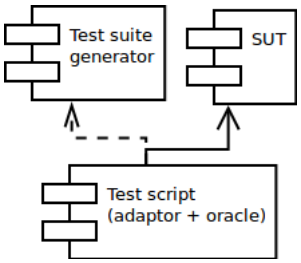

Figure 3: Architecture of the framework

The engine of the framework is the test suite generator component (Figure 3), that based on graph search algorithms. In the test suite generator both famous graph algorithms, the Chinese Postman and New York Street Sweeper algorithms have been implemented, so it can handle models in the form of undirected and directed graphs as well. The test suite generator realised in standalone a Python module.

The test script includes an adaptor for the SUT and a simple oracle. The test adaptor can handle models of SUTs that models are extended with an event data tag (Listing 1).

Listing 1: Extension of GraphML for the test adaptor

```
<key id="d1" for="edge" attr.name="event" attr.type="string"/>
```

The oracle of the framework simply gets the state of the SUT and compares that to the model's state.

5.1 Testing the framework

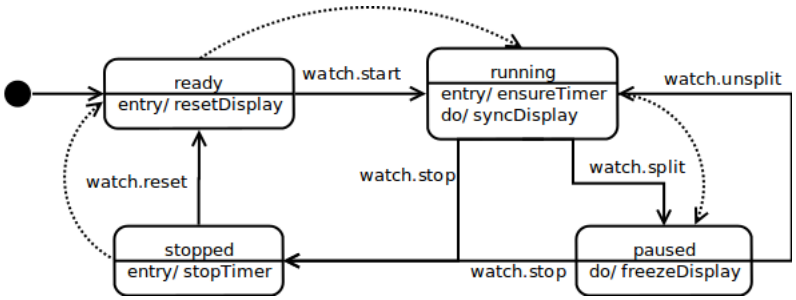

Figure 4: Eulerized UML state diagram of the SUT

To test the framework first a little software has been implemented, that acts like a stopwatch. The UML state diagram of the SUT can be see on Figure 4. The graph representation of this state machine does not have an Eulerian tour. The test suite generator creates a long test case that contains method calls. After the Eulerization of the graph, the Eulerian tour is going to start from the `ready` state, duplicated edges are dotted. The trace of the test execution can be see on Listing 2. In the execution trace one can see, that the methods, which are represented with a duplicated edge on the eulerized state diagram called twice (`start`, `split`, `reset` methods).

Listing 2: Test execution trace

```
TEST0: watch.start
Result: running
PASSED
TEST1: watch.split
Result: paused
PASSED
TEST2: watch.unsplit
Result: running
PASSED
TEST3: watch.split
Result: paused
PASSED
TEST4: watch.stop
Result: stopped
PASSED
TEST5: watch.reset
Result: ready
PASSED
TEST6: watch.start
Result: running
PASSED
TEST7: watch.stop
Result: stopped
PASSED
TEST8: watch.reset
Result: ready
PASSED
```

6 Summary and further development

After investigating the model-based testing theory and its motivation we could see the advantages of this technology. The process of MBT and a possible taxonomy was described, which gave the base for a systematic tools review. Some new properties of MBT related softwares has been defined from a practical point of view. Two concrete testing framework have been presented regarding the previously described properties. Conclusion of the review can be utilised in later works.

Connection to graph theory is a cardinal part of test case generation technologies. The most usable algorithms and some variants have been discussed in that section.

Finally the work has been summarised with a creation of a complete model-based testing framework. The usability and the operation have been demonstrated with a living software.

Finding and demonstrating new technologies, algorithms can be part of the further development.

References

[1] 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.

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

[3] Jonathan Jacky. PyModel. `http://staff.washington.edu/jon/pymodel/www/`, 2013. [Online; hozzáférés 2014.05.24].

[4] Kristian Karl. GraphWalker. `http://graphwalker.org/`, 2014. [Online; hozzáférés 2014.05.24].

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

[6] 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.

[7] 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.

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