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

May 18, 2015

Contents

Kivonat

...

Abstract

...

Chapter 1

Introduction

1.1 Problem and thesis statement

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.

1.2 Proposed approach

...

Chapter 2

Related work

...

2.1 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 [16] 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 [15] collected the available tools that rely on state-based models and created a systematic review considering the previously and newly defined criteria.

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

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

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

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

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

2.1.1 GraphWalker

The first investigated tool was GraphWalker [12], which can create online and offline tests from finite state machines, extended finite state machines or from both of them. The framework is written in Java, the related tools belong to Java world as well. Maven is used to run the tests, TestNG to describe the test cases.

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

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

2.1.2 PyModel

PyModel [10][11] is an open-source MBT testing framework written in Python. It consists three main tools:

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

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

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

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

2.1.3 Conformiq

Conformiq Designer is one of the most famous, industrial model based testing tool. It is available as a plugin for Eclipse, and in a form of a standalone testing framework. Seeing the success of this software, the design of the software has to be investigated.

The MBT process using Conformiq is identical to the original high level MBT process as it can be seen on Figure 2.1.

1. First step is specifying requirements. Conformiq support a huge amount of industrial requirements modelling tool (eg.: IBM Rational, Rhapsody, Sparx Systems Enterprise, ArchitectHP Quality Center, IBM RequisitePro, DOORS), but it contains an

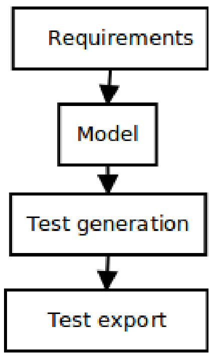

Figure 2.1. MBT process in Conformiq

own editor too. The defined requirements are traceable through the whole software testing process.

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

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

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

2.1.4 GOTCHA

...

2.1.5 ParTeG

...

2.1.6 Conclusion

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

- 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.

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

Table 2.1. Summary of examined MBT tools

- Scaffolding solutions of the tools are incomplete. Fully automatic generation of test adaptors, oracles are seldom supported.
- Regression tests are not supported.
- Creators of these tools either try to use an UML like model or FSM (EFSM). FSM models are low level representations of the SUT, so implementing search based algorithms and graph traversal algorithms are relatively easy. When tools using UML model with graph intermediate model, they can not support complex UML state chart elements, such as orthogonal regions.
- The intermediate model is just always some kind of graph representations, because the test case generation algorithms are the easiest to implement using graph models (search based test case generation, coverage criteria).
- Important thing to note, that the most successful tools use symbolic execution and it can handle the most complex models.

Chapter 3

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 [16]. This section introduces the concept of this variant of software testing through a concrete process (Figure 3.1).

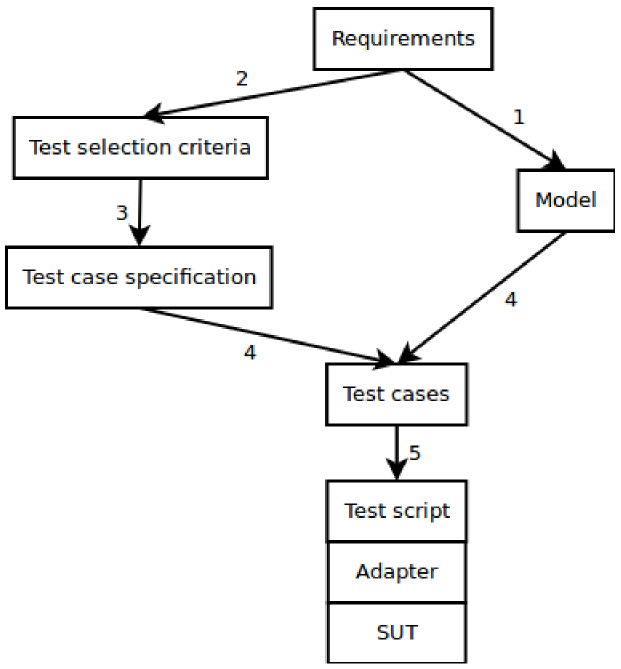

Figure 3.1. Model-based testing process

1. From informal requirements or created specifications a model can be built. The model is an abstract representation of the *system under test (SUT)*. It uses encapsulation to information reduction, because it has to be more simple, than the original system to achieve an easier modifying, maintaining [2]. 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 **??**.

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

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

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

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

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

3.1 Terminology

...

3.2 Taxonomy

Utting, Pretschner and Legeard investigated the currently available MBT solutions and defined (Figure 3.2) a taxonomy which concentrates to three major properties of model-based testing. The three dimensions of their taxonomy are the modelling specification, test generation and test execution.

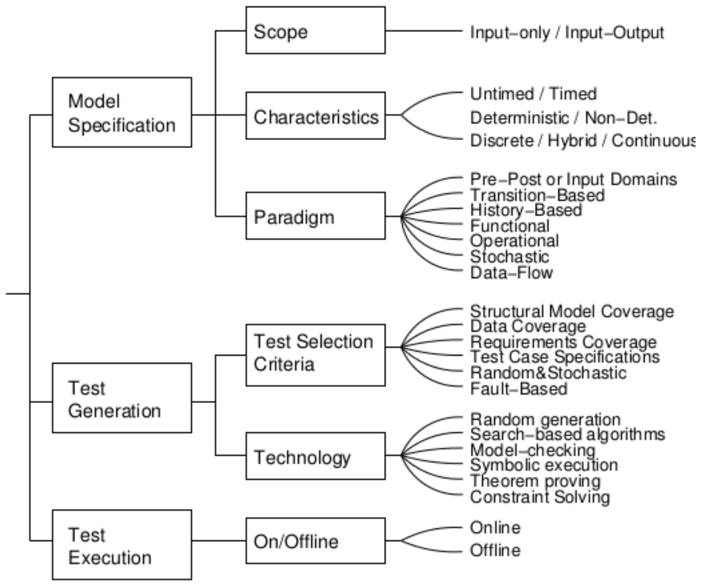

Figure 3.2. Model-based testing taxonomy [16]

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

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

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

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

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

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

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

3.3 Process

...

Chapter 4

Implementation

...

Chapter 5

Scaling and measurements

...

Chapter 6

Summary and further development

...

List of Figures

List of Tables

Bibliography

[1] 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.

[2] 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.

[3] The Eclipse Foundation. Papyrus. `http://eclipse.org/papyrus/`, 2014. [Online; hozzáférés 2014.12.17].

[4] 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.

[5] 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.

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

[7] Conformiq Inc. Conformiq. `https://www.conformiq.com/`, 2013. [Online; hozzáférés 2014.12.17].

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

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

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] Stephan Weißleder. ParTeG. `http://parteg.sourceforge.net/`, 2014. [Online; hozzáférés 2014.12.17].

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

Appendix

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

Web:
`https://...`
Git URL:
`git@...`
Licence:
Apache License, Version 2.0