Observation-Based Modeling for Testing and Verifying Highly Dependable Systems – A Practitioner's Approach

Teemu Kanstrén[1], Eric Piel[2], Alberto Gonzalez[2], and Hans-Gerhard Gross[2]

[1] VTT, Kaitováylá 1, Oulu, Finland
teemu.kanstren@vtt.fi
[2] Delft University of Technology, Mekelweg 4, 2628 CD Delft
{e.a.b.piel,a.gonzalezsanchez,h.g.gross}@tudelft.nl

Abstract. Model-based testing (MBT) can reduce the cost of making test cases for critical applications significantly. Depending on the formality of the models, they can also be used for verification. Once the models are available model-based test case generation and verification can be seen as "push-button solutions." However, making the models is often perceived by practitioners as being extremely difficult, error prone, and overall daunting.
This paper outlines an approach for generating models out of observations gathered while a system is operating. After refining the models with moderate effort, they can be used for verification and test case generation. The approach is illustrated with a concrete system from the safety and security domain.

1 Introduction

Testing consumes a large portion of the overall development cost for a software project. Because testing adds nothing in terms of functionality to the software, there is a strong incentive towards test automation with Model-Based Testing (MBT). Once the models are made and appropriate tools are available, MBT is a push-button solution. Making the models of the System Under Test (SUT), to be used for automated processing and test case generation, does not add any immediate auxiliary value to the final product as well. Moreover, it is typically perceived by practitioners as being difficult, expensive, and overall daunting. One solution for circumventing the difficult and costly manual design and construction process to obtain models for MBT is to generate them out of observations automatically [5], e.g., with the aid of a process mining technique [9].

Obviously, this method of observation-based modeling has to be "bootstrapped" and, therefore, works only on existing software with existing runtime scenarios, e.g., field data and existing test suites [2]. Because most typical software projects in practice have test suites, Observation-Based Modeling (OBM) can be adopted easily by practitioners, and can, eventually, offer automated support for constructing system specification models to be used for system testing following system evolution.

This article presents and outlines a method for model-based testing driven by observation-based modeling. The method is supported by a compilation of existing techniques and tools that have been combined and integrated in order to devise a practical, iterative and (semi-) automatic way to support the creation of behavioural models out of execution traces (observations). The models are made specifically for model-based testing, and they are applied to test and verify a component of a maritime safety and security system. Evaluation of the proposed approach indicates that system specification models for a security system can be boot-strapped from existing execution scenarios, and that they can be refined into models suitable for MBT with relatively little manual user involvement.

The paper is structured as follows. Sect. 2 presents work related, Sect. 3 describes our proposed approach of model-generation, verification, refinement, and testing. Sect. 4 presents evaluation of the work, and finally, Sect. 5 summarizes and concludes the paper with future directions.

2 Background and Related Work

OBM demands that (test) executions of the system under test can be observed, also referred to as tracing. Tracing is widely used in dynamic analysis of programs and it can be applied to observe which components, methods, or basic building blocks are invoked during execution, in order to turn this information into a behavioural model of the software [2]. In addition, external tracing mechanisms such as aspects [6] provide the advantage that the source code does not have to be amended for supporting the tracing.

Finite State Machines (FSM) and Extended FSM (EFSM) are of particular interest for behavioural modeling and, consequently, for behavioural testing [8]. They describe the system in terms of control states and transitions between those states. EFSM add guard conditions to the more general FSM.

Bertolino et al. [1] proposed three steps to the automated "reverse-engineering" of models to be used for model-based testing, but they never realized their proposition. Our method outlined here takes their ideas further and discusses a concrete implementation with existing tools. Ducasse et al. [4] use queries on execution traces to test a system. In this article, we apply similar techniques to help understand what a system does, and to test it. D'Amorim et al. [3] apply symbolic execution and random sequence generation for identifying method invocation sequences of a running system. They devise the test oracle from exceptions and from monitoring executions violating the system's operational profile, described through an invariant model. Our proposed method follows their approach of generating the test oracle. Lorenzoli et al. [7] present a way to generate EFSM from execution traces, based on FSM and Daikon[3]. They use the EFSM for test case selection in order to build an optimal test suite.

[3] `http://groups.csail.mit.edu/pag/daikon`

3 Observation-Based Modeling

Observation-Based Modeling turns the traditional MBT approach around as described in [1]. Instead of creating a model manually, based on a (non-formal) specification, the model is created from the implementation, based on executing a limited number of initial test cases, and tracing their executions. OBM can be used to generate the test model for MBT, the test harness, and the test oracle, by monitoring the SUT's input and output during a selected set of execution scenarios. The entire process can be divided in four different activities, as detailed below.

3.1 Capturing a set of observations

The first step in OBM is to capture a suitable set of observations to be used as a basis for the initial model generation. To obtain observations, the SUT behaviour is monitored while exercising it using a set of existing execution scenarios, such as existing test cases, recorded user sessions, or field data [2].

The main information required to be captured are the *messages passed* through the input- and output-interfaces of the SUT, and the *SUT internal state* each time a message is passed. Typical component middlewares allow to list the component interfaces and to capture all component interactions, without having to instrument every component individually. Obtaining the internal state might be harder, as our approach strives to be compatible with black-box components. Accessing this information typically requires an additional test interface or serialization interface designed into the SUT. In case this is lacking, either the SUT must be manually extended, or it could be possible to maintain an "artificial" state out of the inputs and outputs.

3.2 Automatic generation of the model

The second activity consists in processing those traces and generating an initial behavioural model. This model, expressed as an EFSM, requires the production of states, transitions, and guards.

The generation of the initial EFSM comprises four phases. First, the static parts of the model are generated. These parts are similar for all generated models, and the provided SUT interface definitions are the variables used as input in this phase. Second, an FSM is generated which describes the SUT in terms of interface usage, where each message passed through one of the interfaces matches a state in the FSM. This is done via the ProM tool [9]. This FSM is analysed and processed with specific algorithms to capture the interactions (states and transitions) for the EFSM. Third, invariants over the SUT internal state and parameter data values are provided, and then used to generate constraints, i.e., transition guards, for the interactions and for the processed data values (input data). Finally, all these separate parts of the model are combined to produce the complete EFSM. Fig. 1 presents a very simple example of EFSM specified in the same way as a model generated by our tool. The current state of the model is reported by one special method `getState()`. Every transition is described by one method (e.g.: `vend()`) plus an associated method describing its guard (e.g.: `vendGuard()`).

```
public class VendingMachineModel implements FsmModel {
  private int money = 0;

  public Object getState() { return money; }
  public void reset(boolean b) { money = 0; }

  @Action public void vend() {money = 0;}
  public boolean vendGuard() {return money == 100;}

  @Action public void coin25() {money += 25;}
  public boolean coin25Guard() {return money <= 75;}

  @Action public void coin50() {money += 50;}
  public boolean coin50Guard() {return money <= 50;}
}
```

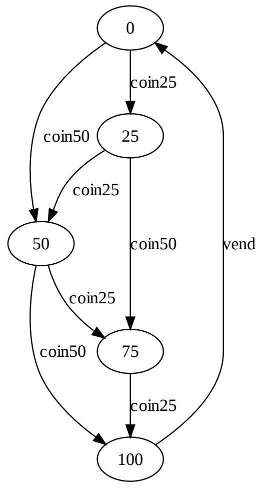

Fig. 1. Example EFSM of a vending machine.

3.3 Test execution

In order to generate the test cases out of the EFSM, our approach relies on ModelJUnit[4]. A test case is created for every possible path going through the various states, along the transitions. Let us note that in this type of model, the lack of some states or transitions compared to the "perfect" model signifies only that the modeled behaviour is not complete, but still represents only allowed behaviour. It is therefore possible to run the test execution before the model is finalized.

In our approach, each transitions contains code to actually send and listen messages from the SUT. Each transition also contains JUnit[5] assertions to determine if the correct messages were answered. The triggering of an assertion implies failure of the test case. The test case is considered passed if no assertion was triggered during the entire execution of the path.

3.4 Manual refinement

The fourth activity for the MBT consists in manual improvement of the generated EFSM. It is typically performed *in parallel* to the test execution activity.

[4] `http://czt.sourceforge.net/modeljunit`
[5] `http://www.junit.org`

In addition to defining the initialization of the complex variables, the task of the engineer is to refine and generalize the EFSM to match the generic expected behaviour of the SUT, which might be different from the observed behaviour. This manual activity should be done in little gradual steps, guided by the results of the tests which exercise the new paths introduced by the generalization of the model at the precedent step.

4 Case study

An example SUT called Merger is used to illustrate the techniques. It is part of a maritime surveillance system. It receives information broadcasts from ships called AIS messages and processes them in order to form a situational picture of the coastal waters. The Merger acts as temporary database for AIS messages, and client components can consult it for track information of ships, or receive notifications of certain ship events. The SUT is also used by software controlling the main screen in the command and control centre for displaying ship tracks. The system comes with a specification in plain English defining behaviour and communication protocols of its components. The components are implemented in Java, executed under the Fractal component framework[6].

4.1 Qualitative evaluation

The Merger component was first instrumented to allow observing a few variables representing its global state and the method calls. Then, observation of the component was performed while 6 manually written unit tests were run and during five minutes of normal operation with field input data. An EFSM was generated out of the traces. This model was manually refined (470 lines of code were changed over 1700) mainly by defining initialization code, generalizing the guards, and correcting the expected behaviour depending on the specification. The refinement process took place with feedback from the generated tests which gradually tested more behaviour of the component. When a bug in the Merger component was found, it was immediately fixed and the refinement process resumed. The refinement and testing process was finished when all the states were accessed, and none of the generated test cases failed.

During this process several errors were found. These errors can be classified into three main types: mismatches between implementation and specification (3), ambiguities in the specification (1), and problems in the design that cause errors under certain conditions specific to the testing environment (2). Overall, in terms of identifying previously unknown errors of a component that had been used for some time in this context, this can be regarded as a very successful model-based testing experiment with real value to the quality of the system.

The evaluation of the method presented, performed in the Merger case study, indicates that the models generated can be used well for model-based testing after moderate manual amendments.

[6] `http://fractal.ow2.org`

4.2 Quantitative evaluation

In order to evaluate the efficiency of the approach, two quantitative evaluations have been also performed. First, using the source code of the Merger component, the coverage of the generated tests has been measured. The measurements are shown in Table 1. Here, *Unit tests* refer to the six initial unit tests used as execution scenarios. *EFSM* refers to the tests generated by the MBT tool out of the final refined model. The four columns correspond to four different types of coverage: statement, method, conditional, and path coverage. This latter one is the number of unique paths in the final EFSM which were followed during a test.

Source	Statements	Methods	Conditionals	Paths
Unit tests	53.5%	64.5%	38.7%	6
EFSM	64.1%	67.7%	48.4%	87
EFSM + Unit tests	65.5%	67.7%	51.6%	92

Table 1. SUT coverage breakdown by execution scenarios.

It is visible that the tests generated from the model provide a significant increase in coverage over the used unit tests. The EFSM set outperformed the initial tests by a small percentage due to observation of the system also on field data, as well as due to the generalization of the generated model in the verification and testing refinement phase. This generalization permitted execution of additional parts of the code, while most parts executed by the original tests are still executed. The biggest difference is in the Paths metric. EFSM largely outperforms the initial tests. Nevertheless this is what is to be expected from an MBT tool that is intended to generate complex interactions to test the SUT.

Second, mutation testing was used to evaluate the effectiveness of the generated test suite. Mutation testing consists in introducing a modification in the code of SUT, and to check whether a test suite is able to detect this “mutation”. 117 “mutants” were automatically generated, of which 51 were considered semantically equivalent after manual inspection. The results are shown in Table 2. When a test finds no errors (the SUT is considered to operate fine), the result is termed “positive”, and oppositely, when an error is reported, it is termed “negative”. “False positives” are the mutations reported fine although it was manually verified that they behave outside of the specification sometimes. “False negative” would be a case where a correct SUT is classified as having an error.

Source	True positive	False positive	True negative	False negative	Total
Unit tests	51	16	50	0	117
EFSM	51	15	51	0	117

Table 2. Mutation test results.

The final model provides minimal gain over the initial unit tests. The model outperforms the unit tests in correct categorization of mutants with actual modified behaviour only by a slight margin. Nevertheless, it is worthy to note that the correct categorizations done by the EFSM are a superset of the one performed by the unit tests. The generated model could detect all the bugs originally detected by the units tests and more.

5 Conclusions

Dependable systems need a high quality of engineering in order to ensure the stability and the correct behaviour of the implementation. Models are useful assets for system engineering. They can be used for specification, verification, reasoning, and test case generation. Once models are available, powerful tools and techniques can be applied to support a range of activities. However, making the models is still perceived by practitioners as being difficult, costly, and error prone. A way to circumvent the difficult modeling process is to have specification models derived automatically from observations from a running system. Because such models specify observed behaviour, rather than expected behaviour, they have to be amended, in order to be applied, eventually, for verification and test case generation.

This paper has presented a approach to bootstrap, refine, and verify models from execution traces, to be used primarily for model-based testing.

References

1. A. Bertolino, A. Polini, P. Inverardi, and H. Muccini. Towards anti-model-based testing. In Fast Abstract in The Int'l. Conf. on Dependable Systems and Networks, DSN 2004, Florence, 2004.
2. B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering, 2009.
3. M. d'Amorim, C. Pacheco, D. Marinov, T. Xie, and M.D. Ernst. An emprical comparison of automated generation and classification techniques for object-oriented unit testing. In: 21st Intl. Conf. on Automated Software Engineering (ASE'06), pp. 59–68, Tokyo, Japan, Sept. 2006.
4. S. Ducasse, T. Girba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In: Conf. on Software Maintenance and Reengineering (CSME'2006), pp. 37–46, 2006.
5. A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In 9th European Conf. on Software Maintenance and Reengineering (CSMR'2005), pages 112–121, 2005.
6. G. Kiczales, E. Hilsdale, J. Hugumin, M. Kersten, J. Palm, and W. Griswol. Getting started with AspectJ. Communcation of the ACM, 44(10):59–65, 2001.
7. D. Lorenzoli, L. Mariani, and M. Pezze. Automatic generation of software behavioral models. In 30th Intl. Conf. on Software Engineering (ICSE'08), pp. 501–510, Leipzig, 2008.
8. M. Uttig and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufman, 2006.

9. W. M. P. van der Aalst, B. F. van Dongen, C.W. Gnther, R. S. Mans, A. K. A. de Medeiros, A.Rozinat, V. Rubin, M.Song, H. M. W. Verbeek, and A. J. M. M. Weijters. Prom 4.0: Comprehensive support for real process analysis. In Application and Theory of Petri nets and Other Models of Concurrency 2007, volume 4546, pages 484–494. Springer, Berlin, Germany, 2007.