Constructing Test Behavior Models Using Simulated System Answers for the Analysis of Test Behavior Anomalies

Benjamin Zeiss[1], Andreas Ulrich[2], and Jens Grabowski[1]

[1]Software Engineering for Distributed Systems Group, University of Göttingen
{zeiss,grabowski}@cs.uni-goettingen.de

[2]Siemens AG Corporate Technology
andreas.ulrich@siemens.com

Abstract: In the standardization of test specifications, it is common that no actual systems exist against which the tests can be executed. Test specifications are developed abstractly in high level languages such as the *Testing and Test Control Notation* (TTCN-3), but they can only be executed when a separate adaptation layer is implemented. Static syntactical and semantical analyses as provided by the compiler and proper manual code reviews are the only means to find mistakes in such test specifications at early stages of design. In this paper, we demonstrate that it is possible to execute abstract test specifications when the system does not exist yet. We use the information provided within the test cases to simulate answers of the system by generating inverse messages to expected messages in the abstract test case. By following a specific coverage-criterion strategy, we are able to execute a sufficient amount of test paths to reverse-engineer behavioral models of test cases which can then again be used for the analyses of potential problems.

1 Introduction

In the last few years, the complexity and size of test specifications developed by standardization and industry have increased dramatically. While there has always been a strong focus on the quality of such software and how to achieve high quality standards, little attention has been paid on the quality of the software products that measure the quality of the software: the software tests. In modern development processes and standardization, there is no often useful means to execute tests. The *System Under Test* (SUT) is either not finished yet or the test specifications are developed in an abstract manner. Mistakes are often not found until an actual execution against the SUT can take place. In addition, the analysis whether a fail scenario occurs due to a fault in the test specification or due to an actual error in the SUT can be costly.

Obviously, it is sensible to find as many mistakes in test specifications as possible early on. This can be accomplished by manual and automated code reviews. There have been numerous approaches to automated quality assessment of software in general. They include static analyses, dynamic analyses, or translations to input languages

of model checkers [CGP00]. Some interesting examples of such works are, for example, [CDH+00] and [Hav99]. In addition, the release of Martin Fowler's book on refactoring [Fow99] raised the awareness of bad smells in code, i.e., code constructs that indicate quality problems. In comparison, there is little work regarding the quality of test specifications despite the fact that they are just as error-prone and have a direct influence on the quality of the software that is tested. Among the published work on quality of test specifications are the identification and adaption of software metrics, bad smells in test code as well as test-specific refactorings in the context of TTCN-3 [Mes07, NZG+08]. Furthermore, a quality model for test specifications has been proposed [ZVS+07]. Most of the involved smell detections in test specifications so far, however, are based on static analyses. One way to achieve a more dynamic type of analysis is to create behavioral models of the tests. With the assumption in mind that the tests cannot be executed against an existing SUT, the first intuitive reaction is that these models can only be generated by static analysis techniques, i.e., analyses that do not execute the test. But this is only partially true: test specifications using message-based communication, for example, contain enough information about the SUT — in particular which messages they expect as stimuli and which answers are supposed to be the correct ones — to actually allow test executions without the SUT. We first outline a new method to reverse-engineer abstract test behavior models from test executions against a simulated SUT that is using and tracking this implicit and internal information by means of test specification instrumentation. Then, we analyze these models regarding structural properties using model checking techniques.

2 Reverse-Engineering of the Test Behavior Model and Analysis

Our method is tailored for the application with TTCN-3 [Eur07] which is a test specification language standardized by the *European Telecommunications Standards Institute* (ETSI). We expect that the basic concepts are adaptable to different languages and platforms though. The overall method is based on the following idea: a test should always be deterministic and automated. Thus, the only ways the behavior of a test can be influenced are: the answers from the SUT and test suite parameters (i.e. module parameters in TTCN-3). Based on this observation, we notice that we are able to steer the execution of the test case into any reachable behavior if we are able to control the SUT answers. To achieve this kind of control, we instrument the test case specifications to log exactly the information necessary to generate such artificial SUT answers that cover the test behavior and reconstruct a model representing the interprocedural control-flow as well as structural events that may happen during the test execution. Rather than just tracing observable messages, we track all the data that we need to reverse-engineer the model.

2.1 Test Behavior Model

To check behavioral properties of tests, an appropriate test behavior model must first be defined that contains the necessary information about these properties and that is suitable

for the representation of the test behavior. A model often used for the representation of reactive systems is the *Labeled Transition System* (LTS). We adapt the LTS to include variables and guards. It is influenced heavily by the *Extended Finite State Machine* (EFSM). Essentially, a transition is only enabled when a guard predicate evaluates as true and a transition may change the value of a variable. The number of variables and what exactly they represent depends on the property that we want to analyze on the model.

The adapted LTS is a more compact representation of a transition system where data does not form a part of the state space and operations on data are modeled through the variables and the transitions. The actual semantical state space by means of the conventional LTS representation could be much bigger or even infinite depending on the value range of the variables as the overall set of states would be represented by the cartesian product between states and the sets of the possible values of each variable. With this in mind, a finite and compact representation is more convenient while it may still be unfolded to a normal LTS. We assume that multiple of those models can be interleaved with a synchronous composition operator. For the asynchronous communication paradigm, we model queues as recursively defined models of the same type with limited recursion depth for bounded queues and infinite recursion depth for unbounded queues. Most of these definitions and semantics are adapted from what can be found in common literature on formal methods and concurrency (e.g., [MK06]).

In our model, each state only represents a distinct control flow position where the data values may have an arbitrary configuration. We do not attempt to model the entire type system of TTCN-3 within our model. In that case, we would be replicating large parts of the TTCN-3 semantics. Thus, we restrict the variables to be of the boolean type that model structural events. The motivation for this abstraction is the purpose of this model: to check properties on the model, we use boolean atomic propositions that we insert on instrumentation already. The drawback of this approach is that we are abstracting from variables and data of the original behavior and thus we will get false positives in the analysis results. So a manual inspection and evaluation of the analysis results is always necessary. Not all language concepts of TTCN-3 have a direct counterpart in this model and have to be partially accounted for by the method that is doing the reverse-engineering of the model. The model semantics merely represent a sufficient subset.

2.2 Reverse-Engineering the Model from Log Data

The first step in the reverse-engineering approach is a determination of what must be logged during simulation. This is essential as it must contain all the information that is necessary to reconstruct our behavioral models. The most significant parts that need to be logged are the *process id*, an *event identifier*, *scope start* and *scope end* as well as the messages. The *process id* enables the creation of separate models for each process. Tracking the history of each *scope start* and *scope end* allows us to put each event identifier into its unique behavioral context. Figure 1 depicts the reverse-engineering methodology. It is a fully automatic and iterative process that terminates if a given coverage criterion is met (here: branch coverage of the test code). To generate the log tuples, we need to instrument

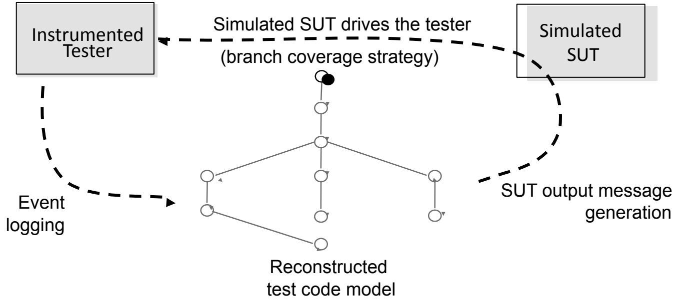

Figure 1: Reverse-Engineering Methodology

the test code to dump the events. In the simplest case, we would log those events after each statement. The log of each simulated execution is processed to construct an increment of the reconstructed test code model which is successively completed in subsequent iterations. To complete a partial model of the test code, SUT output messages are generated such that other execution branches are entered in the model. Visited branches are marked to guide the simulation of the test run.

A challenge in this approach is the generation of the right messages. The problem can be regarded as a test data generation problem. For our first practical experiments, we create the messages by using a rule-based generation of random data based on the available message descriptions and the involved type information. We believe that this approach works reasonably well for most practical cases since branching in typical TTCN-3 test cases due to the receipt of different SUT output messages do not exceed about five or six cases typically.

2.3 Analysis of the Test Behavior Model

So far we have collected about ten problems that can be analyzed on the reconstructed models. Since we are abstracting from data and variables of the test behavior, we analyze only structural properties at the moment, for example, if certain events happen after another event took place, certain event orders, or similar. A consequence is that a model M' which is created by removing all variables from a model M inherits more behaviors than M and thus safety properties satisfied by M' are also satisfied by M. The reason for this effect is that variables effectively are used to forbid the execution of transitions by the use of guards that are not fulfilled. Removing those guards thus directly leads to more behaviors.

The analysis itself takes place by converting the sequential models of each process to Promela, the input language of the Spin model checker, and by mapping the queue and composition semantics to Promela. The description of the properties that are subject of the analysis is realized using *Linear Temporal Logic* (LTL) which is also the standard description of temporal properties using Spin. The applied LTL formulas are often similar and

have an underlying specification pattern as presented by Dwyer et al. [DAC98]. By applying model-checking to our parallel interprocedural models, we are able to analyze also those behavioral paths that are not executed through the branch coverage criterion and we are able to analyze event orders across different processes, i.e., we account for all possible interleavings. Among those problems that we analyze are issues like inconsistencies with the test verdict, for example, a negative verdict is set prior to any communication with the SUT, connection violations according to the TTCN-3 standard, test components that are created, but not started, altsteps that are activated, but not deactivated, or send/receive on stopped/halted ports.

2.4 Discussion

The central points of discussing the approach are the following: First, we suspect that the actual test data generation for the inverse messages may not always be easy. There have been plenty of test data generation methodologies presented in the past [Edv99], but in general, it is not a problem that can be solved cheaply. We suspect though that the generation of the inverse message might not be as troublesome in some cases. There are typically only a handful of alternative messages provided when a message is expected. So the scope of the data generation is somehow limited. Second, we currently abstract from data and variables within the test that also may have influence on the behavior. By applying data abstraction, we effectively introduce more behavior in our reconstructed models than we actually have in our original test behavior. With this choice, we accept that our analysis will probably report false positives though. To model the behavior more completely without such restrictions, we would need to map a bigger part of the TTCN-3 semantics to our model and symbolically log variable manipulations, variable guards, etc. as well. This would drastically increase the complexity of the whole approach. In general, we prefer to keep the complexity of the theoretical model low. Finally, by using branch coverage as target coverage criterion, we attempt to keep the number of necessary executions for the model reconstruction low. We strongly believe that the coverage criterion can be reduced even further with some adaptions to the method and model.

3 Conclusion

We outlined a method to reverse-engineer a test behavior model on which we analyze structural properties of the test behavior. The approach is based on logging events from instrumented test code and using these logs to reconstruct partial model increments that can be directly reused to generate artificial SUT answers based on generated inverse messages to the expected ones. The novelty of the approach lies in the reconstruction of a behavioral model by actually executing the test without an existing SUT. Afterwards, we use model checking to analyze potential faults in the model.

We are currently implementing the approach and plan to apply it to test cases of industrial size test suites. As the instrumentation is different for each property to be analyzed, we

consider to use aspect-oriented techniques to apply the instrumentation. We expect that the approach can be automated to a comprehensive extent where only the LTL formulas and a declarative description of the instrumentation need to be specified manually. However, we expect that the actual generation of the inverse messages may present a problem on some occasions and that the described approach may need to be refined. Finally, we plan to incrementally revise the approach to reduce the abstraction degree and compare the results with the complexity and effectiveness of an approach that constructs the same model by using only static analysis without real test executions.

Acknowledgements

The work is financially supported by Siemens AG Corporate Technology and we like to thank Testing Technologies for continuing tool support.

References

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models From Java Source Code. In *International Conference on Software Engineering*, 2000.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[DAC98] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceeding of the 2nd Workshop on Formal Methods in Software Practice (FMSP)*. ACM, 1998.

[Edv99] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, 1999.

[Eur07] European Telecommunications Standards Institute. *ETSI ES 201 873-1 V3.2.1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, 2007.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Hav99] K. Havelund. Java PathFinder, A Translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking*, Lecture Notes in Computer Science (LNCS), page 1680, 1999.

[Mes07] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[MK06] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. Wiley & Sons, 2nd edition, 2006.

[NZG+08] H. Neukirchen, B. Zeiss, J. Grabowski, P. Baker, and D. Evans. Quality Assurance for TTCN-3 Test Specifications. *Software Testing, Verification and Reliability (STVR)*, 18(2), 2008.

[ZVS+07] B. Zeiss, D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski. Applying the ISO 9126 Quality Model to Test Specifications. In *Proceedings of SE 2007*, volume 105 of *Lecture Notes in Informatics (LNI)*. Köllen Verlag, 2007.