Dissertation VTT PUBLICATIONS 727

Teemu Kanstrén

A Framework for Observation-Based Modelling in Model-Based Testing

VTT

VTT PUBLICATIONS 727

A Framework for Observation-Based Modelling in Model-Based Testing

Teemu Kanstrén

Academic Dissertation to be presented, with the assent of the Faculty of Science, University of Oulu, for the public discussion in the Auditorium IT115, Linnanmaa, on February 19th, 2010, at 12 o'clock noon.

ISBN 978-951-38-7376-9 (softback ed.)
ISSN 1235-0621 (softback ed.)

ISBN 978-951-38-7377-6 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

Copyright © VTT 2010

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 5, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

Edita Prima Oy, Helsinki 2010

Teemu Kanstrén. A Framework for Observation-Based Modelling in Model-Based Testing. Espoo 2010. VTT Publications 727. 93 p. + app. 118 p.

Keywords model-based testing, test automation, observation-based modelling, test generation

Abstract

In the context of software engineering, test automation as a field of research has been around for a very long time. Yet, testing and related concepts are still generally considered to be one of the most time-consuming and expensive parts of the software life cycle. Although it is a field with a relatively long research background, many existing test automation systems are still relatively simple and not very different from the early days. They still focus on executing an existing, usually manually crafted, set of tests over and over again.

One approach that has also been around for a relatively long time but has only recently begun to attract considerable interest in the domain of software testing is model-based testing. In model-based testing, the system under test is represented by a model describing its expected behaviour at a higher abstraction level, and a set of chosen algorithms are used to generate tests from this model. Currently, these models need to be manually crafted from the specification.

This thesis presents an approach for observation-based modelling in model-based testing and aims to provide automated assistance for model creation. This includes design and architectural solutions to support observation and testing of the system, analysis of different types of executions used as a basis for observations, and finally combines the different viewpoints to provide automated tool support to generate an initial test model, based on the captured observations, that is suitable for use in model-based testing. This model is then refined and verified against the specification. As the approach reverses the traditional model-based testing approach of going from specification to implementation, to going from implementation to specification, guidelines for its application are also presented. The research uses a constructive approach, in which a problem is identified, a construct to address the problem is designed and implemented, and finally the results are evaluated.

The approach has been evaluated in the context of a practical system in which its application discovered several previously unknown bugs in the implementa-

tion of the system under test. Its effectiveness was also demonstrated by generating a highly complete model and showing how the completed model provides additional test coverage both in terms of code covered and injected faults discovered (test mutants killed).

Preface

Once upon a time I was young(er) and started geeking around with computers and programming at an early age. I still enjoy the nostalgia of watching recordings from these times on YouTube. From somewhere along the line I picked up my desire to understand and analyse program behaviour and its different properties. This is also reflected in this thesis where the underlying theme is really satisfying my desire to understand how programs work and what makes them work that way.

This work was carried out together with different institutions and with financial support from a number of sources. Part of the work was conducted as a member of the Automated Testing Platforms Team in the Software Architectures and Platforms group at the VTT Technical Research Centre of Finland, with funding from different projects at VTT. Part of the work was also carried out in the context of the University of Oulu, with funding from the SoSE Graduate School on Software Systems and Engineering. The final parts of the thesis were finished during my research visit to the Software Engineering Research Group at the Delft University of Technology, the Netherlands. I have also carried out some of the work in my own time and in my own place, while I was financially supported by scholarships from the Jenny and Antti Wihuri Foundation, and the Nokia Foundation, and while wearing on the nerves of my family.

I wish to thank all the above institutions for making it possible for me to work on interesting research topics and to create this thesis.

I wish to thank my supervisor Ilkka Tervonen and the reviewers of the thesis, Per Runeson and Atif Memon. I also wish to thank the people who made my visit to TU/Delft both interesting and possible, Arie van Deursen and Hans-Gerhard Gross. I also thank my parents for all their support over the years, whatever I have chosen to do (or not) and whether it made sense or not. Finally, I thank my wife Kaisu and my children Valtteri and Severi (almost the cover art-

ists). Any simple list would not do you justice, so thank you for everything. Too bad Pikku-Pupu and Niinan tyyny did not make it to the thesis as I hoped.

I dedicate this thesis to the memory of my brother Jari, who once helped me get back to software engineering and was there to see me start the journey for a PhD but never to finish it.

The Hague, December 2009 Teemu Kanstrén

List of original publications

I Kanstrén, T., Hongisto, M. & Kolehmainen, K. 2007. Integrating and Test-ing a System-Wide Feature in a Legacy System: An Experience Report. Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR'07, Amsterdam, the Netherlands, 21–23 March, 2007. 10 p.

II Kanstrén, T. 2007. Towards Trace Based Model Synthesis for Program Understanding and Test Automationm. Proceedings of the 2nd International Conference on Software Engineering Advances, ICSEA 2007, Cap Esterel, French Riviera, France, 25–31 August, 2007. 10 p.

III Kanstrén, T. 2008. Towards a Deeper Understanding of Test Coverage. Journal of Software Maintenance and Evolution: Research and Practice, JSME, Vol. 20, No. 1, 2008. Pp. 59–76.

IV Kanstrén, T. 2008. A Study on Design for Testability in Component-Based Embedded Software. Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications, SERA'08, Prague, Czech Republic, 20–22 August, 2008. 8 p.

V Pollari, M. & Kanstrén, T. A Probe Framework for Monitoring Embedded Real-Time Systems. Proceedings of the 4th International Conference on Internet Monitoring and Protection, ICIMP 2009, Venice, Italy, 24–28 May, 2009. Received best paper award and invitation for extended version to the Journal on Advances in Systems and Measurements. 7 p.

VI Kanstrén, T., Piel, E. & Gross H.-G., Observation Based Modeling for Model-Based Testing. Submitted to the Journal of Software Testing, Verification and Reliability, 2009.

VII Kanstrén, T. Behavior Pattern-Based Model Generation for Model-Based Testing. Proceedings of the 1st International Conference on Pervasive Patterns and Applications, PATTERNS 2009, Athens, Greece, November 15–20, 2009. 9 p.

VIII Kanstrén, T. Program Comprehension for User-Assisted Test Oracle Generation. Proceedings of the 4th International Conference on Software Engineering Advances, ICSEA 2009, Porto, Portugal, September 20–25, 2009. 10 p.

Contents

Abstract 3

Preface 5

List of original publications 7

List of abbreviations 11

1. Introduction 12
   - 1.1 Research context 14
   - 1.2 Research questions 16
   - 1.3 Research approach 17
   - 1.4 Contributions of the thesis 25
   - 1.5 Structure of the thesis 26

2. Test automation and observation-based modelling 27
   - 2.1 Test automation 27
     - 2.1.1 Terminology and basic concepts 28
     - 2.1.2 Test data generation 32
     - 2.1.3 Test oracles 40
     - 2.1.4 Test harness 44
     - 2.1.5 Model-based testing 45
   - 2.2 Observation-based modelling 47
     - 2.2.1 Basic terminology 49
     - 2.2.2 State-based models 50
     - 2.2.3 Other models 51
   - 2.3 Discussion 54

3. A framework for observation-based modelling in model-based testing 56
   - 3.1 Phase 1: Defining the target model 56
   - 3.2 Phase 2: Applying the framework 59
     - 3.2.1 Step 1: Capturing observations 60
     - 3.2.2 Step 2: Model generation 62
     - 3.2.3 Step 3: Model refinement for verification and testing 63
   - 3.3 Discussion 64

4. Introduction to original papers 66
4.1 PAPER I: Integrating and Testing a System-Wide Feature in a Legacy System: An Experience Report 66
4.2 PAPER II: Towards Trace-Based Model Synthesis for Program Understanding and Test Automation 67
4.3 PAPER III: Towards a Deeper Understanding of Test Coverage 68
4.4 PAPER IV: A Study on Design for Testability in Component-Based Embedded Software68
4.5 PAPER V: A Probe Framework for Monitoring Embedded Real-Time Systems 69
4.6 PAPER VI: Observation Based Modeling for Model-Based Testing 70
4.7 PAPER VII: Behavior Pattern-Based Model Generation for Model-Based Testing 71
4.8 PAPER VIII: Program Comprehension for User-Assisted Test Oracle Generation 72

5. Framework evaluation 74
5.1 Study subjects 74
5.2 Phase 1: Defining the target model 76
5.3 Phase 2: Applying the framework 76
5.3.1 Step 1: Capturing observations 76
5.3.2 Step 2: Model generation 77
5.3.3 Step 3: Model refinement for verification and testing 77

6. Conclusions 79
6.1 Answers to the research questions 79
6.2 Limitations and future work 80

References 83

Appendices

Papers I–VIII

List of abbreviations

AJAX	Asynchronous JavaScript And XML
AMBT	Anti-Model-Based Testing
COTS	Commercial Off-The-Shelf
CP	Category Partition
DFT	Design For Testability
DOM	Document Object Model
EFSM	Extended Finite State Machine
FSM	Finite State Machine
GUI	Graphical User Interface
HTML	HyperText Markup Language
IO	Input/Output
MBT	Model Based Testing
OSM	Object State Machine
SUT	System Under Test
SW	Software
UML	Unified Modeling Language
V&V	Verification and Validation
WS	Web Service
WSDL	Web Service Definition Language
XML	eXtensible Markup Language

1. Introduction

Software testing is one of the activities used to ensure the proper implementation and functionality of a given software artefact. It can be described as “observing a software system to validate whether it behaves as intended and identify potential malfunctions” [1]. As it is rare to write software without trying to run it and thus implicitly trying to test it, software testing can be considered to be as old as the writing of software. All things considered, however, life is not that simple and someone who is into, for example, formal methods may disagree in the spirit of the famous quote from Donald E. Knuth, “Beware of bugs in the above code; I have only proved it correct, not tried it.” [2]. Regardless of how one looks at it, software testing as a research area has a long history, at least in the context of software engineering. Gelperin and Hetzel [3] date the first articles on program testing to the 1950’s and the first conference on software testing to 1972. They also describe the evolution of software testing over different periods. More recently, software testing has been seen as encompassing all parts of the development and maintenance processes and as something that needs to be included and planned for from the beginning [1].

Software testing research is commonly justified by arguing that testing takes up to 50% or more of the total development costs of software (e.g. [1]). As this is usually based on decades-old studies (e.g. [1]), the validity of this reasoning is arguable. Most of the discussion on the matter seems to agree that testing is one of the most costly parts of the software development process, however, so there is bound to be some truth in it. In software testing research, this is typically used to emphasize the importance of related research, and this thesis is no different. This emphasis alone on the cost of software testing makes software test automation research important from the cost-effectiveness perspective of software engineering.

From the application domain perspective, the ever-increasing complexity of software-intensive systems and their increasingly pervasive nature also high-

lights the importance of research into software testing (automation). In the last few decades, software has become a commodity with a growing presence in everything around us. This includes the "simple" things such as the electronic toothbrush, complex systems of systems, and life-critical applications in healthcare and other domains. In this context, it is also increasingly important for software to fulfil its required purpose reliably. This in turn emphasizes the need for better support in techniques, including test automation, to verify the correctness of SW systems.

The term "test automation" seems to imply much: automated testing of a given software system. One might think that given any system, the automated testing platform would test it and verify its correctness. In practice, however, most existing test automation platforms focus on automatically running existing test scripts that first have to be manually created. This way, they do little more than allow for the repetition of existing test cases, which is also often referred to as regression testing. As well as having to write the test scripts manually, the user of these platforms also typically has to take care of other tasks such as connecting the test cases inside the test platform to the SUT. This part is referred to as the test harness.

Some approaches take automation further. These will be discussed in more detail in Chapter 2. One of these approaches is model-based testing in which an MBT tool generates test cases based on a model of the SUT. This model typically describes the SUT at a higher level of abstraction than the implementation, usually as a black box focusing on its external interfaces and higher-level functionality [4]. In this way, MBT tries to automate more of the testing process, as the model is expected to require less effort to maintain than a manually created suite of test cases. This can still require significant effort, however, in creating and maintaining the test models, including acquiring the specialist skills for producing good models suitable for MBT.

The research presented in this dissertation aims to make the process of MBT more cost-efficient and easier to adopt. It presents tools and methods to automate much of the process of generating a test model for a SUT. It discusses the breaking down of a chosen target model into a set of observations (e.g. traces of program execution such as messages passed through external interfaces) that can be used to automatically generate an initial model suitable for MBT. An implementation to automatically generate an extended finite state-machine model from a given set of suitable observations is provided and used to evaluate the concept. The generated model is not intended to be perfect as is but to provide an ad-

vanced starting point from which to quickly start the MBT testing process and help to model the behaviour of the SUT accurately while verifying the correctness of the implementation against the specification.

The usefulness of this type of approach has already been discussed by Bertolino et al. [5], who discussed this type of concept but never implemented or studied it in practice. This approach reverses the concept of model-based testing, which typically uses the system specification as a basis for the manual creation of the test model. This model is then executed and compared against the actual implementation. Now, instead a model is automatically generated from the implementation and, with the aid of an MBT tool, used to verify the implementation against the specification following a set of provided quidelines. The generated model combined with an MBT tool includes everything needed to generate and run tests against the SUT, including the test scripts (for SUT input), a test harness and test oracles (to verify the correctness of SUT output).

1.1 Research context

Software testing is considered to be part of the software verification and validation processes. Together, V&V aim to make sure that the software delivers what the customer expects from it [6]. Verification focuses on assessing whether the software matches its specification, and validation on assessing that the specification matches the customer's needs. SW testing is a tool in these processes for ensuring better SW quality. V&V also include many other tools and techniques, such as inspections and static analysis methods [6]. Similarly, SW testing can be defined from many perspectives.

The definition and focus of test automation used in this thesis follow the definition given in the software engineering book of knowledge, which defines software testing as consisting of "the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior." [7].

In addition, testing is often divided into categories, such as functional and non-functional testing, white-box and black-box testing, and different levels of testing such as unit- and system-level testing. While much of the research presented in this thesis can be applied to the context of many of these different types of tests, the main focus of this thesis is on a subset of these. This chosen subset is described next.

Functional testing focuses on the functional properties of the software, such as correct input-output transitions. Non-functional testing focuses on the non-functional attributes of the SW, such as performance and usability. The research presented in this thesis focuses on testing the functional properties of SW. The tests generated by the described test automation framework are functional tests for the SUT. However, many of the described results are also applicable to some non-functional requirements such as monitoring and analysing the SW behaviour for performance analysis [8].

The division of tests into white- and black-box tests is related to the information on which the tests are based. White-box testing makes use of information about the internal structure and implementation of the SUT, whereas black-box testing views the SUT as a black box and focuses on testing through its external interfaces. Of course, the reality is never this black and white; there are gray areas in between where the information can be considered to partly require knowledge of the internal structures and implementation while still being of a higher level. In many cases, the two approaches can also be combined. The research presented in this thesis focuses mostly on a black-box approach, with some gray-box elements such as relying on implemented test interfaces to access the internal state of components.

When defining the research context and approach applied to this thesis, it is also important to define the assumptions made about the SW being tested. Two concepts commonly applied to the design of modern SW systems are the use of components and of services. These are used to define conceptual units of deployment where parts of the required functionality are provided by the different units (as components and services). A common term related to these is the use of service-oriented architecture (SOA). While no strict limitations are presented, the research in this thesis is oriented towards SW systems to which this type of structuring approach is applied. Many of the publications also address embedded real-time systems specifically. These define the scope of the analysed SW systems, although the presented tools and techniques are not seen as being restricted to these domains.

The concept of different levels of testing is discussed in more detail in [9]. In the context of this thesis, the focus is on testing components and services, mainly with a black-box approach as described earlier. In this regard, no distinct classification is made as to which levels of testing are addressed, as this depends on various definitions such as what constitutes a unit, a module, a component or a service that is being tested. The research described in this thesis, however, is more

oriented towards the higher levels of testing of components and services as opposed to considering very small units of code such as single methods or classes.

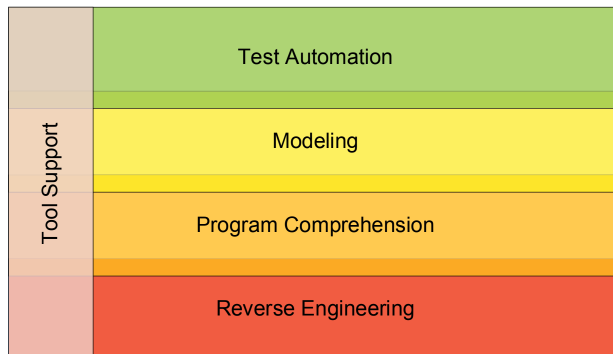


Figure 1. Research context.

The research presented in this thesis combines approaches from different software engineering research domains as illustrated in Figure 1. Reverse engineering-related concepts are used to capture observations (information) about the SUT behaviour. Automated tool support is used to turn these observations into models suitable for test automation. The set of observations required to capture a comprehensive model are analysed with the help of program comprehension techniques. The program comprehension techniques are also used to further analyse the models to support the user in the process of modelling and using the models for verifying the implementation vs the specification. Finally, automated model-based testing tools are used to generate more tests and to assess the correctness of the SUT implementation against its specification. All parts of this process are supported with software tools.

1.2 Research questions

So far, the process of creating models for model-based testing has been mostly a manual process. In this thesis, means are sought to automate as much as possible of this process. As the focus is mainly on using observations from the execution of an existing system as a basis for this, the main research question can be summarized as follows:

- How can automated support be provided for model creation for model-based testing of existing software?

This question has been divided into a number of smaller subquestions which, when answered, provide a basis for answering the main research questions. These subquestions are defined as follows:

1. How can the information to generate the models be captured?

In order to generate the models, a set of observations first needs to be defined that can be used as a basis to describe the SUT at a level that enables the use of algorithms to generate the target models. The answer to this question requires a definition of the information that needs to be captured, where it should be captured from and how it is to be accessed by the algorithms used to generate the model.

2. How can this information be turned into a meaningful test model for MBT?

The generation of a model from the captured information (observations) requires a set of algorithms that can be used to turn the observations into a model usable for MBT. The answer to this question will provide these algorithms. This question is also closely tied to the first subquestion, as defining the algorithms also defines the information they need to operate.

3. How can the generated models be used for SUT verification and testing?

When a model is generated automatically for a SUT based on observations of its execution, this model accurately describes the SUT as it is. To verify the correctness of the implementation, however, the actual implementation needs to be compared with the expected implementation (as expressed by the SUT specification). As the model is further used to generate tests for the SUT, it should not describe what the SUT actually is but rather what it should be, i.e., the model should be made to reflect the (correct) expected specification when used for testing. The answer to this question will reveal how the generated model can be used to verify the correctness of the SUT behaviour.

1.3 Research approach

Software engineering research applies many different types of research methods. For example, Glass et al. [10] list 22 different types of research methods based on their review of the software engineering research literature. By far the most popular of these are conceptual analysis (conceptual analysis 43.5% and conceptual analysis/mathematical 10.6%) and concept implementation/proof of concept

(17.1%), followed by laboratory experiments (3.0%) [10]. Runeson and Höst [11] classify the different research methods further into four categories:

- Exploratory – finding out what is happening, seeking new insights and generating ideas and hypotheses for new research
- Descriptive – portraying a situation or phenomenon
- Explanatory – seeking an explanation of a situation or a problem, mostly but not necessarily in the form of a causal relationship
- Improving – trying to improve a certain aspect of the studied phenomenon.

Going into all possible research methods in detail is outside the scope of this thesis, and thus the focus here is to present the parts that are relevant to the study in this thesis.

Runeson and Höst describe case studies in software engineering as often taking an improvement approach [11]. In a similar way, this dissertation applies mainly a constructive research approach in which the problem is first analysed and a conceptual framework presented, then an artefact is designed and constructed to address the problem, and finally the results are evaluated [12, 13].

This type of constructive research can also be referred to as design science, which is generally defined as attempting to create things that serve a human purpose [14], or more specifically in the context of information systems as seeking "to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, and use of information systems can be effectively and efficiently accomplished" [15]. In the context of software engineering, this can be translated into the specific properties of software engineering such as tools, methods and processes that support software engineering activities.

In addition to design science, the second main research approach applied to this thesis is that of conceptual analysis. Although listed as one of the main approaches applied to both software engineering [10] and computer science in general [16], it is difficult to find a meaningful definition of conceptual analysis in the context of software engineering or computer science. Neither of these studies [10, 16] provides any explanation for or reference to the definition of contextual analysis, although they define it as one of the main approaches. In this thesis, the definition of conceptual analysis follows the concept of the definition of analysis in [17], while applying it in the context of software engineering research.

In light of this definition, conceptual analysis is defined here as the search for the definition of a given concept and the act of breaking that concept into more elementary parts. Once the individual parts that constitute the definition of the decomposed concept have been defined, the definition and the concept of the whole can be discussed. For example, in this context the definition of test automation is approached by splitting it into its constituent parts (test input, test harness, test oracle, etc.) and using this as a basis for approaching complete solutions for supporting test automation. This definition is thus based on the concept of analysis as discussed in [17].

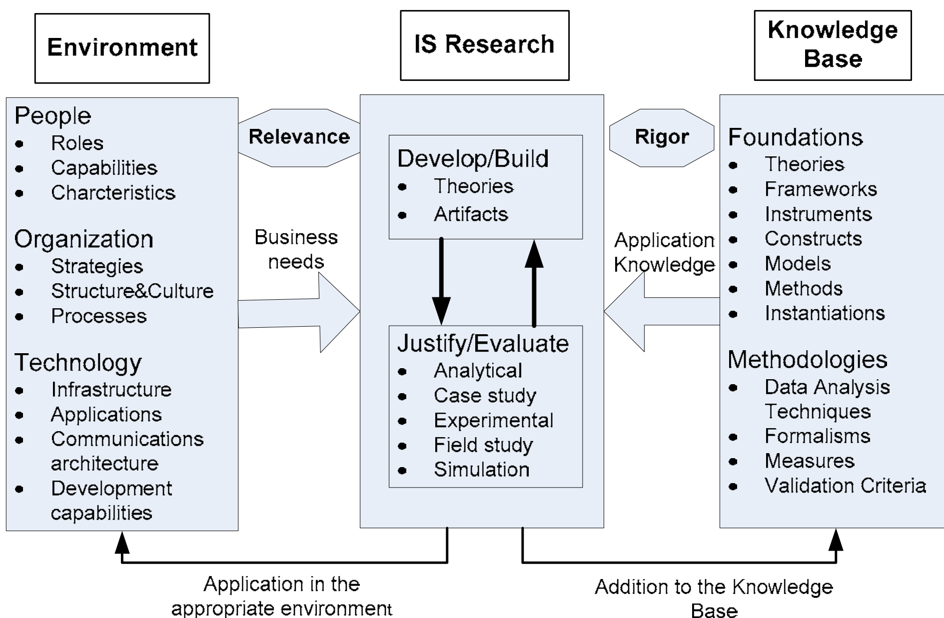


Figure 2. Information Systems Research Framework as described by Hevner et al. [15].

Figure 2 shows the information systems research framework as described by Hevner et al. [15]. In this framework, the environment defines the problem space in which the phenomena of interest reside. Together, the different parts of the environment define the business need or "problem" as perceived by the researcher. Given these business needs, the research is conducted in two complementary phases as shown in the middle of Figure 2. These phases are the building and evaluation of the artefacts designed to meet the identified business needs. The knowledge base provides the raw materials from and through which research is accomplished. Prior research and results from reference disciplines

provide the foundations for the develop/build phase of a research study. Methodologies provide guidelines used in the justify/evaluate phase. Rigour is achieved by appropriate application of existing foundations and methodologies. A link back from the research to both the environment and the knowledge base is present in the research being applied to the actual environment and by providing new knowledge into the knowledge base.

In line with the constructive research approach of design science, the research approach applied in this thesis can be described as a three-step process, progressing through conceptual analysis, artefact construction and case studies presented in the attached papers. Thus this approach also follows the most popular research methods in SW engineering as described by Glass et al. [10] (conceptual analysis and concept implementation). The guidelines for design research as given in [15] are mapped to the different papers presented in Table 1 of this thesis.

In view of the evaluation part of the research approach described in Figure 2, a number of different approaches are possible [18, 15, 11]. The ones discussed further in this thesis are presented briefly below. A more thorough review of the different papers, their study subjects and how they contribute to the evaluation of the overall framework presented in this thesis will be given in Chapter 5.

- A case study is described as typically focusing on what is happening in the context of a single project [18], although it is not uncommon to study several projects [11, 18]. Based on different sources, Runeson and Höst [11] describe case studies as “an empirical method aimed at investigating contemporary phenomena in their context”.
- Controlled experiments can be described as studying the effects of manipulating one variable on another variable, with the requirement that subjects are chosen at random [11]. Kitchenham et al. [18] refer to these as formal experiments, stating also that the study must be replicated several times.
- A survey involves collecting information across many teams and projects with the help of techniques such as interviews, for which the selection of subjects is planned [11, 18].
- Action research, which is similar to a case study but also involved in the change process and aims at evaluating the effect of the change [11]. This approach is not used in any of the publications but is considered for possible future work in Chapter 6.

One additional concept that is relevant to this thesis (and visible in Figure 2) is simulation, which is used to provide synthetic data for the experiments [15]. In the context of software testing this includes the use of artificially injected faults into the system (termed mutation testing) to assess the fault detection effectiveness of a test automation approach. This is used as one of the evaluation techniques for the OBM approach in Paper VI.

Table 1. Design science guidelines [15] and mapping them to this thesis following [19].

Guideline	Description
1: Design as an artefact	A viable artefact in the form of a construct, a model, a method or an instantiation is produced. In this thesis the main artefact is the OBM tool and related process. In addition, an independent artefact that contributes to the whole of the OBM approach is presented in the papers that apply a constructive approach.
2: Problem Relevance	Technology-based solutions to important and relevant business problems are developed. The relevance of each problem is motivated in each paper, and for the complete OBM approach in Chapters 2–4 of this thesis. Paper I also provides descriptions of the general relevance of the research presented in many of the papers.
3: Design Evaluation	The utility, quality and efficacy of a design artefact must be rigorously demonstrated via well-executed evaluation methods. Each part of the whole that constitutes the OBM approach has been evaluated in a realistic and relevant environment as presented in the papers.
4: Research Contributions	Provide clear and verifiable contributions in the areas of the design artefact, design foundations and/or design methodologies. This thesis as a whole presents a new approach to model-based testing, and each paper contributes to a subfield in this area in a more detailed and also independent way (a wider contribution).
5: Research Rigour	Rigorous methods are applied in both the construction and evaluation of the design artefact. Each part of the research that is presented in individual papers has been validated by carefully planned individual studies, including the complete approach in the final paper (VI).
6: Design as a search process	Available means are utilized to reach desired ends while satisfying laws in the problem environment. Each part of the presented research has been iteratively refined during the course of the research. This is most visible in Paper II, which provides some early experiments for OBM that have been taken much further in Papers VI–VIII and Paper IV, which provides insights into the research described in Paper V. In addition, Paper VI combines all the parts, as will be described in Chapter 3.
7: Communication of research	Research presented effectively to both technology-oriented and management-oriented audiences. The technical details of the research have been thoroughly described in the attached papers. In this introductory part, especially in Chapter 3, a more practical overview description is given.

In addition, Runeson and Höst [11] provide a set of guidelines for case studies. They discuss a case study protocol that should formulate the case study plan and contain at least the following elements:

- Objective – The objective defines the initial focus point and may evolve during the study. It may be, for example, one of the four types of research described earlier: exploratory, descriptive, explanatory or improving. [11]
- Research questions – These provide the definition of what needs to be known in order to fulfil the objective of the study. [11]
- The case – This defines what is being studied. For example, it may be a SW development project, a process or a product. [11]
- Theory – This defines the frame of reference for the study. Since theories in SW engineering are not well developed, it can be based on, for example, study of existing methods or the viewpoints taken during the study. [11]
- Methods – This defines how data are collected for the study. Examples include interviews or tool instrumentation. [11]
- Selection strategy – This defines the selection of the studied case, that is, from where the case study data are sought. [11]

Table 2 describes the different papers in this thesis for some of these viewpoints. The selection strategy was based mainly on the available industrial systems for study. As most of the case studies were carried out in collaboration with industrial partners in research projects, this has set the context of the study and the case selection. In these cases, the motivation to use industrial projects has been both the drive for more collaboration with industrial partners and the provision of a realistic environment for the research. A notable exception is Paper III, which uses a freely available open source project as a case study subject. In this case, the choice was based on the properties such as available test cases and project complexity, and the aim was to have an extensible choice of tests and significant complexity available to enable a realistic research context. Paper VI can also be seen to have some elements of a controlled experiment when artificial faults are injected and the fault detection effectiveness of the OBM approach is evaluated.

The most dominant research approach has been that of conceptual analysis, followed by constructing design artefacts and evaluating the improvements gained from its application. In this case, the motivation has been to first obtain a

complete overview of the theory (fundamentals) and current state of the subject area. This is seen as providing an effective research approach in first building an understanding of the constituent parts of the subject (such as test automation) and what has currently been done in the area. From this, new constructs for the chosen goal have been designed and evaluated. This has also been an iterative process, as described earlier. The final overall goal was to build a system for more advanced support of testing through observing and controlling a system, although this has progressed from more simple solutions (Paper II) to more advanced and complete solutions (Paper VIII).

Table 2. Research approaches in the papers in this thesis following [19].

Paper	Objective	Research question(s)	Research approach
I	Describing experiences of testing and analysis of complex, embedded real-time SW	What issues exist in testing and analysis of modern software-intensive systems?	Experience report, descriptive case study on a development project & system
II	Improving regression testing via automated model generation	How can a set of observations be used as a model for regression testing?	Construction, improving, case study on testing an existing system
III	Improving existing test coverage measures and their analysis	What types of tests are needed for different parts of tested SW?	Conceptual analysis, construction, improving, case study of a SW product
IV	Exploratory analysis of design for testability in industry	How can support for testing and analysis be effectively built into a system?	Interviews, exploratory, survey on several projects and teams in 2 companies
V	Providing support for improving behaviour monitoring and testing in SW systems	How can effective support for testing and analysis be built into SW-intensive systems?	Construction, improving, case study on analysis of several SW products
VI	Improving automation of model creation in model-based testing	How can EFSM test models be generated and used? How useful are generated models in practice?	Conceptual analysis, construction, improving, case study on several components in a SW system
VII	Providing guidelines for algorithm development for test model generation	How can algorithms be designed to generate executable test models from captured observations?	Conceptual analysis, construction, improving, case study on one SW system
VIII	Improving automated support for test oracle generation	How can automated support be provided for the creation of test oracles?	Conceptual analysis, construction, improving, case study on SW system

Overall, conceptual analysis is applied to identify relevant problems in test automation research. This is presented in Paper I and in Chapter 2 of this thesis. A set of constructs and related evaluations are presented to address the research problems presented in this thesis. Paper II presents the basic concept of this thesis, which relates to both conceptual analysis and design of the construct. Papers III–V present designs and implementations of constructs to address issues identified in Papers I & II. The remaining papers present the final pieces of the puzzle and provide the constructs to complete the work while also combining this with the constructs presented in the previous papers and providing an evaluation of the results. Paper VIII is a return to conceptual analysis in which the end result is analysed and a framework is presented with some of the research results described in a wider context. The timeline for the progress of this thesis is shown in Figure 3, with each paper being shown on the timeline as P:*X*, where *X* is the number of the paper.

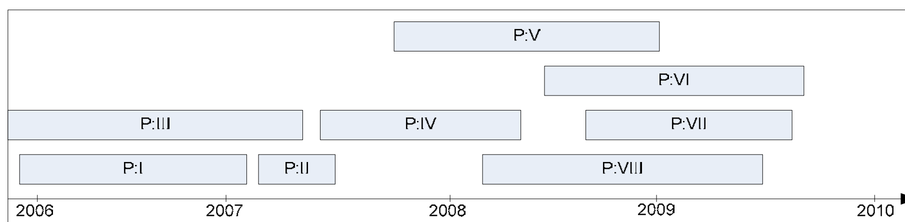


Figure 3. Research timeline.

The review of the state of the art described in Chapter 2 was conducted at the beginning of the work and updated throughout for each of the papers, and finally for this dissertation. Paper I describes experiences from an industrial project on which the author was working from the end of 2005 to 2007. Paper III is a continuation of the author's master's thesis. The remaining papers describe work in research projects after the industrial project described in Paper I. Most of this research work can be related to the experiences described in the industrial experience paper, and most of it was sequential, though work for Paper V was carried out in parallel with a master's thesis worker under the supervision of the author. The work in the last three papers is closely related and thus interleaved. The concept analysis in Paper VIII was conducted for related work before Papers VI & VII and finalised after them.

1.4 Contributions of the thesis

The main contribution of this thesis is the presented automation framework for supporting model-based testing. The main contribution is formed from the original papers that make up this thesis. Each of these papers presents work on different properties needed for the realization of the framework and provides a validation through a case study for the topic of that paper. The thesis covers the related background work in Chapter 2, providing an overall view of test automation and observation-based modelling from the viewpoints relevant to this thesis.

The focus is on techniques related to dynamic analysis, i.e., the analysis of the behaviour of software systems based on observations made from their actual executions. The papers that form this thesis focus on the different aspects of this type of test automation. These papers describe the following properties related to implementing this type of test automation platforms:

- An analysis of the complex environment and the requirements it sets for test automation platforms in a modern software-intensive system.
- Tool and architectural design solutions and guidelines for providing effective testability support for capturing the base observations used for analysing the SUT behaviour.
- A classification framework for and analysis of the different types of executions used as a basis for the analysis of SW behaviour.
- The decomposition of a model for the information to be observed for model generation.
- Support for analysing the set of observations provided, for their completeness in providing a sufficiently complete model of the SUT behaviour.
- Methods, tools and algorithms to turn the captured observations into models for use in model-based testing.
- Experiences and guidelines for using these models.

Based on these contributions, a complete framework for supporting the automation of different aspects of testing modern software-intensive systems with the aid of model-based techniques is presented. The different properties of the test automation platform, as described, are combined to form the framework as presented in the last few papers of this thesis. The framework is summarized in Chapter 3 of this thesis.

1.5 Structure of the thesis

The rest of this thesis is structured as follows. Chapter 2 provides an overview of existing research on test automation and observation-based modelling. First, it describes the different parts of a test automation platform and the state of research relating to these parts. As test automation research is a very large area of research, the focus is on providing a general overview of the field.

Secondly, Chapter 2 also provides an overview of what is called observation-based modelling. This refers to the generation of models describing SW behaviour based on information captured (observations) by monitoring the execution of the SUT. It provides a general overview of different types of models and a more focused overview of the state-based models used in the work presented in this thesis.

Chapter 3 presents the developed framework for generating models based on the observations captured from the SUT execution. It describes the decomposition of a target model to define the information that needs to be captured, how this information is turned into a model usable for MBT tools and the process of using the models with the MBT tools in SW testing and verification. It also shows how the papers composing this thesis relate to different parts of the proposed framework.

Chapter 4 provides a more detailed summary of the papers that compose this thesis.

As the research progressed through various projects and study subjects, Chapter 5 describes the study subjects used at the different phases of the study (in the different papers) and how each of the studies presented in the different papers contribute to the evaluation of the presented framework for observation-based modelling in model-based testing.

Finally, Chapter 6 concludes the thesis by describing how the research questions were answered and discusses the limitations of the work and the need for future research.

2. Test automation and observation-based modelling

This chapter gives an overview of research related to this thesis in the areas of test automation and observation-based modelling. It starts with an overview of the test automation research area, followed by an overview of research in the area of observation based-modelling. Finally, the chapter concludes with a short discussion on positioning the work presented in this thesis in relation to these two fields of research.

2.1 Test automation

The basic form of test automation is that of regression testing in which existing test cases are automatically (re-)executed. These tests can be created manually or with different degrees of automation, and their execution is triggered by some event such as a user pressing a button or committing to version control. A test automation system requires different components depending on its type, such as test scripts, input data, test oracles, a test driver and a test harness. These basic components are illustrated in Figure 4.

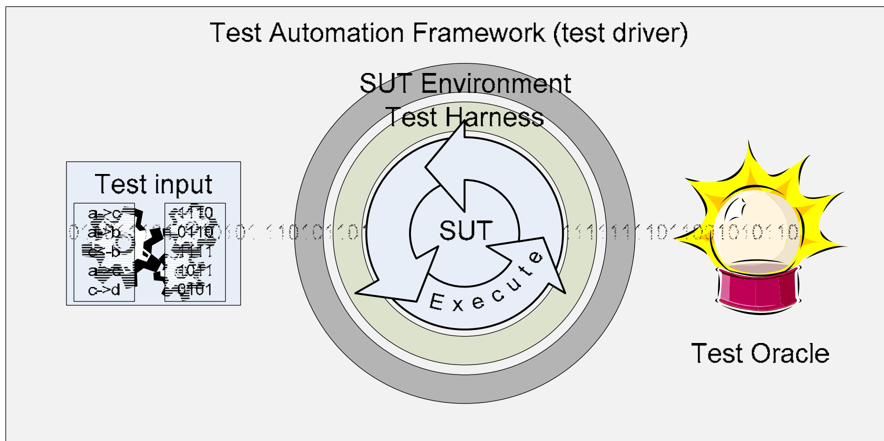


Figure 4. Components of a test automation framework.

A test driver controls the overall test execution process, including the execution of the SUT. The SUT is isolated, for different testing purposes, from parts of its environment (other components or systems with which it interacts) with the help of a test harness. Test input can take different forms, such as message sequences (test scripts) and data values. Generators can also be used to automatically generate large quantities of different types of data. A test oracle is used to verify the correctness of the received output (such as data values or message sequences) in relation to a given input. In the context of this thesis, the SUT is considered as a black box that takes some input and produces some output, with only limited insight into its internal processes.

This chapter reviews the different approaches in test automation research that have focused on the different parts of these test automation platforms. As the amount of available research in this field is vast and constantly growing, the intention is not to provide a complete overview of all related work but rather a comprehensive overview of different parts.

2.1.1 Terminology and basic concepts

Design for testability

Effective implementation of test automation requires certain properties from the system under test (SUT) as well as the test environment [20][21]. The design of the SUT and the test environment architecture must consider the test require-

ments in addition to the program (user) requirements. In short, this is called design for testability (DFT). Two basic terms related to this are observability and controllability. Following Binder [22], controllability can be defined as being able to control the SUT input (and internal state), and observability as being able to observe its output (and internal state).

Experiences have shown that it is especially important to take DFT into consideration from the early phases to enable efficient and cost-effective implementation of test automation [20][23]. While the testability of the SUT itself can be considered from a number of different perspectives [22], from the test automation perspective, the architecture is typically the most important part [20][21].

Techniques to support testability in the SUT architecture include isolating parts of the SUT for testing, accessing information about the system behaviour and providing test functionality [20][21][24]. From the test environment perspective, DFT includes automated creation of system configurations for testing, test interfaces between the SUT and the tester application, abstraction of test models and automated change management between the test cases and the SUT [20][21].

To isolate parts of the SUT for testing, it must be possible to replace clearly defined parts of the SUT with test implementations, commonly referred to as test stubs. Test stubs can be used to isolate parts of the system and to provide test functionality, such as input generation [21][24]. To enable effective partitioning, the interfaces must be separated from the implementations so that stubs can be attached to these interfaces [24].

To access information about the system behaviour, specialized interfaces must be available to read and write data values to program variables [24]. As not all of these are necessarily needed for normal functionality, extra support is often necessary [20][21][24]. This support can be simple interfaces to access data such as system states [24][25] or test functionality such as keeping track of resource consumption [21]. Component internal test support functionality is seen as particularly important in testing components for which no source code is available [26]. The presentation of information is also seen as important in the regard that it still needs to be interpreted by a human analyst at some point, and it has to be effectively processed by the test automation system [20][21].

Invariants

As defined in the first chapter, in the context of this thesis testing is considered as observing and asserting a set of controlled executions (test cases) on a SUT. In this context, techniques designed for this type of analysis are also relevant and often used for testing. One such popular technique that is also used in the context of this thesis is that of dynamic invariant detection. Invariants in general can be considered as properties that hold at different points of analysis, such as over time. The concept of invariants, as discussed in this thesis, is generally related to what is defined as dynamic invariant detection aimed at detecting likely program invariants by Ernst et al. [27]. They define an invariant as a property that holds at some point in a program, and present a tool called Daikon[1] to automatically infer these from a set of program executions [27].

Examples of these invariants include a variable always being constant (x = 1), non-zero (x ≠ 0), variable relations (x < y), a variable being a value of a function (x = fn(y)) and a data structure always being sorted. The invariants are called likely invariants as they only hold for the analysed program executions. The basic output of invariant detection is a model of the system describing it in the form of likely invariants. Likely because they are based on a set of executions of the code, and this set may not include all possible executions. This model can be used for many purposes, such as understanding the system, debugging and test generation [27].

Mutation

When automated testing techniques are developed or automated test cases generated, their effectiveness needs to be evaluated. This requires the availability of faulty software to test the effectiveness of the techniques in a controlled environment [28][29]. Two basic techniques for this are fault seeding and program mutation. Fault seeding is a process of manually adding faults into the program code. Program mutation is a process of applying an automated tool on program code to create mutants [30]. Each mutant is a changed version of the program, and the task of the test suite is then to kill these mutants, i.e., it should identify each mutant as a failing test case. In this case, a measure of the effectiveness of a generated test suite can be the amount of mutants it kills, and this can be used as

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

a basis for test data generation algorithms [31][32]. Studies indicate that mutation is an effective tool for test automation validation purposes, but requires special consideration [28].

There are three requirements to kill a mutant: reachability, necessity and sufficiency [32]. The mutated statement must be reached by the execution of the program to kill the mutant, as the mutated statement is the only changed statement in the program. It is necessary for the execution of the mutated statement to cause an observable change in the program state in order to distinguish a failed (mutated) test run from a correct (non-mutated) run. To be sufficient, the final state must be propagated through the program execution in order to be visible in the test output so the test case can observe the mutant.

Test coverage

A key concept of testing any system or software is to have a measure of how good the current level of testing is. In the context of software testing, this is referred to as test coverage. A basic measure for this is a ratio of which parts of the SUT are executed (covered) by the existing test suite and which parts are not. This can give, for example, a measure that 50% of the SUT code is covered by the tests while the other 50% is not. A basic use for this is then to look at the uncovered 50% and write tests to cover more of this previously uncovered code. As the implementation of features in a SW system often overlaps and features such as error-handling behaviour can be difficult to cover, this is not a straightforward task. Different types of tests, such as unit tests, integration tests and system tests have different roles in testing the SUT, and it is not always meaningful to consider them as a single coverage measure [9]. For different goals, it can also be useful to combine different measures instead of focusing on them separately [33]. In practice, however, resource limitations such as time and money set constraints on how much testing is cost-effective to implement.

To address the issue of resource limitations when executing existing test suites, various techniques have been presented, including regression test selection, prioritization and minimization [34]. These share a goal of optimizing the test suite execution based on a given criterion, such as new tests covering the biggest possible parts of the previously uncovered parts of the SUT.

Coverage measures have also been applied in other contexts, such as mapping program features to their implementation [35] [36] and finding the causes of failures [37] [33]. In these cases, it is also possible to discuss coverage in gen-

eral, as these techniques make use of any program executions and their related coverage information. Thus the executions do not have to be tests (other options include generated data and captured user sessions), although automated tests are a popular basis for this analysis. The analysis techniques that make use of coverage measures typically analyse the complete set of executions and relate their coverage against each other with goals such as locating feature implementations or causes of failures.

Different types of coverage measures exist for test coverage. Examples include measures based on program structure, its data value space, requirements and dynamic properties of its execution [4]. Structure-based metrics measure the coverage of properties such as lines of code and execution paths taken. Data-based measures consider the possible values of SUT input and output and measure how many of the possibilities have been covered. Requirements-based coverage associates test cases against SUT requirements to ensure all requirements are tested. All these measures are based on different forms of static information about the SUT. In addition, measures exist based on dynamic information about the execution of the SUT. Examples include generating mutants and observing how many of these are identified (killed) by the test suite [38], and inferring an invariant model from the test cases to describe the coverage as an invariant model [39].

The different coverage measures show how exhaustive testing of any non-trivial SW system is not practical as it is not cost-effective. In this regard, it is also expensive to create numerous test cases manually to cover all the different aspects needed to verify a SUT to a reasonable degree. For these reasons, test-generation techniques focusing on these different aspects are important and have been the focus of much research in software engineering.

2.1.2 Test data generation

The basic form of a test case is to give the SUT some input and observe its output. A SUT typically has several input interfaces, which accept many types of data. The combinations of the different inputs and their effects on the SUT behaviour need to be considered, as it is not possible to exhaustively test every possible combination. As manually crafting a good input data set for this is difficult and time-consuming, a lot of research has dealt with generating test data for input. This section provides an overview of approaches taken in this field.

Basic approaches

A basic approach in this field is random test data generation. In this approach, randomly selected values from the input domain(s) of the program are used as test input. Random test data generation is intuitively simple and one of the oldest test data generation methods [40]. Although intuitively simple, studies have shown that it can also be effective, and it continues to be an active research area [40] [41] [42] [43]. While the basic application is generating numerical values, the generation of objects has also been addressed. Random testing is often combined with other techniques to make the selection of the random values more advanced, such as creating an equal spread of values [42].

Other basic techniques include the use of equivalence partitioning (dividing the set of possible values into sets that cause equal program behaviour) and boundary value analysis (selecting values that are considered likely to cause errors at variable boundaries). Although basic methods have been in use for decades, this is still an active research area. For example, more recently, Beer and Mohachi [44] have studied the combination of basic random data-generation techniques while considering the effects of variable values on each other.

Symbolic execution is a test data generation approach that represents the evolution of variable values over their control-flow paths in relation to the input values of the path [45] [46]. The program variables and their combinations are represented as symbols and the branches in these paths are turned into constraints (sometimes also referred to as path conditions). These constraints are then solved in relation to the input values and used to generate test data that will exercise all the paths of the SUT. This is referred to as constraint solving [31] [47]. Although commonly applied with symbolic execution for path coverage [47][43][48], constraint solving has also been applied with other types of goals such as generating data to kill program mutants [31] and to reach manually inserted assertions [49].

Symbolic execution and constraint solving have some limitations. They are typically applied on a white-box level, which requires access to the source code and poses complexity problems with non-trivial systems, including code size, pointers and arrays [50]. With powerful modern hardware and algorithms, however, they are being used more widely, also in the context of commercial test automation systems [43] [48].

More dynamic analysis methods have been presented to address issues of test data generation based on static analysis. In these methods, the SUT is actually

executed as opposed to only its static structure being analysed, such as source code, and the observations from the executions are used as a basis for further test data generation. These dynamic analysis approaches have some basic limitations, such as requiring a fully executable system, and the possible effects of the execution, such as launching missiles or changing bank account details, need to be considered. A basic approach in this regard is dynamic path analysis [51], which exercises the SUT with actual input values and monitors the control flow. If the execution takes an unwanted path, analysis algorithms are applied to find values that make the execution take the desired path.

Combination testing focuses on the different combinations of the input parameters. Grindal et al. [52] provide a survey on the different techniques in this field. The basic concept is described in the category partitioning method [53], which is the basis of many of these methods. In the category-partition method, the program is first partitioned into functional units that can be tested separately. For each of these, parameters and affecting environment variables are identified, as well as the possible values for each of these individually. Constraints are identified between the parameters, variables and values. Finally, combinations satisfying these constraints are generated for the parameters, and these are transformed into test cases. As it is not realistically feasible to test all possible combinations, different strategies have been applied to find the combinations of interest. These include simple solutions such as including each value once, and more complex solutions such as search-based algorithms [28].

While many of these basic approaches work with primitive object types, their use for the domain of object-oriented programs has also been studied by Thummalapenta et al. [54]. In this case, the problem is in the large possible state space of the created objects that may be used as inputs. Thummalapenta et al. [54] extract object method invocation sequences from existing source code repositories and use these to produce object creation sequences with the goal of reaching a chosen target state. Target states are identified by the coverage measures required, such as achieving higher branch coverage in unit tests that make use of objects of the generated type.

Search-based approaches

Search-based optimization techniques have been widely applied to different fields of software engineering, including test data generation [55] [50]. Search-based optimization aims to optimize a set of data for a given goal using different

algorithms to generate data and the analysis of the results to guide further data generation. The use of these techniques requires two properties of the optimization problem to be defined: a representation of the problem and a fitness function [55] (sometimes also referred to as the objective function [50]). The representation is used to encode the problem into something that can be processed automatically, and the fitness function is used to rank the results to guide future data generation.

In test data generation, the analysed problem is one of generating test data to fulfil a given test criteria. Thus, the possible inputs to the SUT form the search space, and the problem representation must encode these inputs in a way that can be manipulated by the search algorithms. The test criteria are translated into the fitness function. For example, if the goal is to cover a chosen execution path, the fitness function can measure how close the test data come to executing that path, or in the case of finding worst-case execution times, it can measure the time it takes to run a test case [55]. Another example of a more black-box approach is test data generation for a car parking system presented in [56], in which the fitness function is the distance between the car and the collision area, and the representation is the description of the parking environment (car, collision area, parking area).

Traditional search-based optimization techniques applied in test data generation include hill climbing, simulated annealing and evolutionary algorithms [50]. Some problems remain in applying these algorithms as they are best suited to numerical representations, and more complex data structures and internal states of objects are problematic for them [50]. Thus, innovative encoding of the input space is often required. Although search-based test data generation has mostly only considered one criterion, it is also possible to optimize test data generation based on multiple criteria. For example, Harman et al. [57] have optimized for branch-coverage as well as dynamic memory allocation using both the path distance and the amount of memory allocated in a test run as a problem representation.

Some work has addressed input data generation specifically for object-oriented programs as the parameters of method calls for unit tests [58]. In this case, a set of predefined generators is provided to generate any primitive value types, and custom generators can also be provided for any parameter. Object values are generated by applying the value generators to their respective constructor methods recursively as needed.

Combinations of search-based approaches and other test data generation methods have also been presented. Ayari et al. [59] use a search-based test data generation method for killing mutants in which the fitness function is based on

the distance of a generated test case from killing a mutant. In this case, the fitness function is based on the formulas for constraint-based test data generation presented in [31]. Baudry et al. [60] have presented an approach called bacteriological algorithms, which is also based on program mutation, in which the best data set for killing mutants is chosen from each generation, and after this all killed mutants are removed from the data set. In the end they collect all the chosen data sets to function as separate test cases. In this way, it is a search-based approach with a more specific algorithm tuned for test data generation.

Domain-specific approaches

Like the previously described search-based test data generation approach for a car parking system, many forms of domain-specific test data generation have been applied. Examples include considering the domain-specific properties of the program and making use of the available domain-specific formal data structure specifications.

For example, using domain-specific knowledge of the underlying programming language constructs, Bertolino et al. [61] used the category partition method to generate test data from XML Schema descriptions. They created test scripts based on a formal test specification and a number of manual steps [53]. The steps included the identification of functional units in the specification, partitioning these into categories, then these further into choices and finally determining constraints between the choices. Bertolino et al. [61] map the different properties of the CP method to elements of the XML Schema, such as subschemas to functional units and categories to XML element types. Based on this mapping and a provided XML Schema, they generate XML instances for use as test data. Similarly, for web services, Sneed and Huang [62] have used the web service description language as a basis to generate web service invocations. They use random test data generation, but form these into more complex compositional data types based on the analysis of the WSDL that describes the data types of the application.

Yuan and Memon [63] have presented a feedback-based technique for generating test cases for graphical user interfaces (GUIs). The state of a GUI is composed of triplets, which include a widget, its properties (such as colour, size, font) and the possible values of these properties. This state is considered to be changed by discrete events that describe actions and interactions between the GUI elements. A seed test suite is obtained from an event-interaction graph

(EIG), which is created by a GUI reverse-engineering algorithm. By analysing the states and their relationships in these executions, they build up an event-semantic interaction graph to describe these properties. From this, they then generate an event-interaction semantic graph and, finally, use this graph as feedback from the previous test cases to generate additional interaction sequences as new test cases.

Daniel et al. [64] present a technique for generating test cases for refactoring engines. They provide a programming library called ASTGen to produce abstract syntax trees (ASTs) as input for refactoring engines. This approach is partially manual as it supports the building of generators to generate input programs that exhibit desired properties of programs to be refactored and tested. Thus, in this case the input data generation takes the form of manually writing input generators that generate ASTs for SUT input, and this is supported by the framework provided.

Wang et al. [65] describe a technique for automatically generating tests for context-aware applications. They analyse the SUT source code to find context-aware program points (capps) where context changes may affect the program behaviour. Using static analysis, a control flow graph is generated, which describes how the capps affect the program behaviour. Input data are generated in the form of new control flows that cover more context switches between different capps. These take the form of control-flow scripts that are generated to be input according to the way the program should be manipulated to traverse different control flows related to its context-aware behaviour.

Program-invariant-based approaches

Pacheso and Ernst [66] present a technique for automated generation of test cases based on program invariants, which they have implemented in a tool called Eclat. This technique is based on two inputs: the program to be tested and a set of correct executions of the program. They use Daikon to generate an operational model of the program under test. This model is based on a set of invariants inferred from a set of execution scenarios that are expected to describe the correct behaviour of the SUT. They generate input for the SUT aimed at producing behaviour that violates the previously inferred operational model and is considered to be potentially fault revealing. This input takes the form of method calls, with parameter values provided from a pool of input values. This pool is initialized with a few primitive values and a null object. Further values are added, as

returned from method and constructor calls. The results are labelled as normal, faulty or fault revealing. Behaviour is considered normal if both the program inputs and outputs match the program invariants, fault revealing if the input matches the invariants but the output does not, and illegal if neither the input nor the output matches the invariants. Any thrown errors or exceptions also result in a fault-revealing classification. The inputs and outputs considered are the standard instrumentation used by Daikon, input and output values of each method execution.

Agitator is a commercial tool for automating parts of the unit-test generation process [43]. It makes use of various existing test input data generation methods, such as symbolic execution, constraint-solving and feedback-directed random input data generation. Both static and dynamic analyses are used to analyse the different execution paths of the SUT and provide a basis for test data generation. Specialized constraint-solvers are provided for specific constraints of the execution paths, such as Boolean functions of Java String objects. Heuristics are used as an aid to guide test data generation, such as using values -1, 0, 1 as integer input values. More complex objects are modified with mutator methods, and when they cannot be automatically generated, user-specified factory objects are used to generate test data objects. Feedback from executions with given values is also used to improve the input values. The aim is specifically to translate different input data generation methods from research into practically usable forms for a commercial tool, working with code bases of significant complexity. In this regard, they make a number of approximations to optimize the performance of these algorithms. In their case they find that since they need to cover paths many times in order to generate usable invariants, it does not matter if the produced data for the chosen paths are not always 100% exact [43].

Use of field data

Typically, test data for a program are generated during development on development platforms based on assumptions about the way the program will be used and how it will function in actual use. A different approach is to use data collected from the field, from actually deployed systems used by actual users. Elbaum and Diep [67] provide an overview of research in this area and a set of empirical studies on their application. One basic means to generate tests with field data is to include test functionality in the product itself. In this case, the program can be profiled during the development time and the program executions in the field can be monitored with regard to breaking the assumptions in

these models. From a more traditional viewpoint, field test data can be applied to generate test cases to match the user sessions described in the field data, or to augment existing test cases with executions and elements missing but identified in the field data. The field data collected by Elbaum and Diep are collected using program instrumentation. They also study the effectiveness of different instrumentation techniques and find good results with targeted instrumentation that focuses on the relevant properties of the tested functionality.

Test scripts

As mentioned earlier, especially in the context of object-oriented programs, the generation of method-invocation sequences has also been a topic of test data generation. In this case, a set of constructor and method invocations are combined to create new test cases. These are often combined with various test input generation methods as described earlier to produce data such as parameter values.

One example of this is the feedback-directed random testing technique described by Pacheco et al. [68]. In this technique, tests are constructed by randomly selecting constructor and method calls to invoke. Valid sequences are defined by the used contracts, which are objects classifying sequences as valid or invalid. Further sequences are built based on previously generated valid sequences, and this feedback is used to guide the generation of the sequence. Example contracts include an object always equalling itself, and a method not throwing certain platform error exceptions. Filters mark call sequences that violate the contracts as illegal sequences that are not to be used for generating further sequences. As a result, the tool provides two types of tests: failing and passing. Passing tests are sequences that violate no constraints. Failing tests are ones that violate a constraint, and this constraint is described with an assertion included in the test case. In addition to call sequences, they also apply the same analysis to variable values passed to the method invocations. They also describe a successful application in an industrial case study [69].

Tonella [58] has used search-based algorithms to generate both test input data (as described earlier) and method-invocation sequences for unit testing. For the method call sequences, the methods of a class are described along with their potential parameter values. Search-based algorithms are used to generate new sequences of method invocations to test the class under test, including the generation of required parameter objects as described earlier. The aim is to generate tests that satisfy a given coverage criteria.

Briand et al. [25] use constraint-based test sequence generation for COTS components based on component specifications. Possible method invocation sequences are described as 3-tuples of preceding method, succeeding method and predicate. In this notation, the preceding method always has to come before the succeeding method and the predicate further defines at which point this transition can occur. The predicates describe the constraints that are inferred from the component specification. Algorithms are used to solve these constraints and to construct fitting method-invocation sequences.

2.1.3 Test oracles

As described earlier, a basic form of a test case is that of providing some input to the SUT and observing the resulting output. The output then needs to be asserted in order to define if the result was correct and matches the expectations set for it. The component of a test automation system that does this is called the test oracle. This section first discusses the properties of test oracles and then reviews the different approaches to the test oracle problem taken in test automation research.

The terminology relating to test oracles is used according to [70]. A *test oracle* is defined as a mechanism for determining the correctness of the behaviour of software during (test) execution. The oracle is divided into the *oracle information*, specifying what constitutes the correct behaviour, and the *oracle procedure*, which is the algorithm verifying the test results against the oracle information.

Further terms are also used according to [70]. Successful test evaluation requires information to be captured about the running system using a *test monitor*. For simple systems, it can be enough to just capture the output of the system. For more complex systems, such as reactive systems, more detailed information such as internal events, timing information, stimuli and responses need to be captured. All the information captured by the test monitor is called the *execution profile* of the system and it includes control and data information.

The term test oracle may be confusing at first. Who or what is this oracle and what does it have to do with SW testing? It is a fitting choice for what it describes however. An oracle is typically considered to be a mystical source of wisdom and information. In the case of a test oracle, this also holds true. The information on what the correct behaviour and response of the SUT are comes from somewhere (a mystical place). Basically in the case of SW, this information is typically defined in the specification. From the test automation perspective, this is often problematic however.

If the specification is written using a formal notation, it may be possible for a test automation system to automatically read this notation and use this information to define the expectations (the oracle information). In practice, however, most specifications are written in natural language and are often imprecise and incomplete. Thus a test automation system, even when generating different forms of test input, cannot automatically determine in which case the resulting output from the SUT is correct and according to expectations. In fact, if an automation system existed that could define what is expected from any SUT, without any external information, it should then also be able to generate the SUT itself as well as any system that has not even been defined. Thus, it is clear that fully automated test oracle generation without external input as such is not possible.

Most test automation techniques then require the user to provide a manually defined test oracle for the test cases. It is possible to use a generic test oracle to check for crashes in the SUT (unexpected exits or thrown exceptions) [63], however, such approaches do not work when it comes to checking any other type of application-specific output that typically forms the actual test oracles for a SUT. Sometimes very specific test oracles can be provided for a given test automation system [61] [62] [64], but these do not generalize and only partially cover the SUT specification.

In many cases, the oracle procedure is provided and the information needs to be provided. In other cases, however, it is the other way round and the information may be inferred from an existing system but requires the definition of the procedure that correctly analyses this information. Any combination of these is also possible. In any case, the oracle in test automation always requires some form of manual augmentation (such as [71]) or, if fully automatic, is very specific and narrowly applicable. In the case of formal specifications, it is also possible to use these specifications as the oracle information [72]. As described earlier, however, these are outside the scope of this thesis.

Supporting techniques

Many techniques relating to test oracle automation do not provide or generate an automated test oracle themselves, but rather focus on supporting the user in creating the oracle information, the oracle procedure or both. Typically in these cases, the execution profile is captured from the SUT using a test monitor, and algorithms are provided to make assertions based on the execution profile. In these cases, the basic oracle information is usually available and the oracle pro-

cedure needs to be provided. It is then the job of the user to describe the oracle procedure using the tools provided.

Andrews and Zhang [73] have presented a technique for test oracle generation based on log file analysis. This is based on the SUT writing a log file based on using a predefined logging policy, and a log file analyser asserting the correctness of the execution based on the log file. Their approach requires the log file analyser component to be written as a test oracle and provide a matching logging policy to map this oracle to the log file. They illustrate the approach with state-machine-based matching, in which the transitions are based on the available log lines. The log-file analyser component is applied against log files collected from SUT execution and makes an assertion of whether the log file matches the expected behaviour or not.

Both Ducasse et al. [74] and Roover et al. [75] have described similar techniques for building test cases based on traces collected from a program execution. They start by executing the SUT and collecting traces from the execution. Logic languages derived from Prolog are used to query the execution traces, and these queries act as the test oracles. They assert that the recorded behaviour matches the expected behaviour. Ducasse et al. [74] use the queries to filter relevant data from large, low-level data sets, while Roover et al. [75] perform similar queries but also aim to limit the trace data to higher level events and lighter trace implementation. The aim of these techniques is to produce a model that is both humanly understandable and machine verifiable in order to support both test automation and program comprehension.

Program invariants are used as a basis for assisted oracle generation in Agitator [43], Eclat [66] and the technique proposed by Xie and Notkin [76]. All of these provide the user with a set of invariants as inferred from the set of executions using the test input described in the previous section. They then provide the user with the option of turning these assertions into actual test oracles and adding them to the existing test suite for the program.

Automated oracles

Some techniques have been presented to provide domain-specific automated test oracles. These typically come with their own specific input-generation mechanisms and are only applicable to very restricted types of applications. These test systems can then be used to assess the properties of this type of applications. As described earlier, these automatically generated oracles cannot describe the cor-

rectness of any application-specific properties but rather focus on chosen generic viewpoints.

Daniel et al. [64] present an automated test oracle for their testing technique that is intended for automated testing of refactoring engines. They use a number of test oracles such as invertibility of the refactoring operation (performing the operation backwards produces the original result), checking that the refactored code compiles, and specific assertions for chosen refactoring operations such as moving an element actually resulting in creating the item in a new location.

A similar approach has been taken by Mesbah and Deursen [77] who provide test oracles for GUI testing AJAX-based web applications. They use a set of invariants specifically defined for this type of applications as the oracle information. This includes generic invariants such as the HTML output always being valid and the DOM tree not containing any error messages. It also includes application-specific invariants that are (manually) defined specifically for each application such as clicks on page elements updating the displayed table of contents. Their automated test oracle procedure checks that these invariants are not violated during the use of the web application by an automated input-generation tool.

Memon and Xie [78] have developed test oracle information extraction techniques for GUI testing. They use the previously described GUI properties to describe the possible states of the GUI. An execution monitor is used to capture the state of the GUI after each event given to the SUT. They use a set of test cases that are considered to describe the correct behaviour of the SUT to produce a model of the expectations for these states. The produced model then describes the oracle information for regression testing.

Machine-learning techniques are used to automatically learn models based on a chosen set of algorithms from potentially large data sets. As a result, they typically provide an evaluation function that describes the given data in some way. There are two phases to applying most machine-learning techniques. First, a set of training data is given to the classification algorithm to create the model. In this phase, the classification of the data needs to be provided in order for the algorithm to build (learn) a model for the classification. In the second phase, this model and algorithm is applied to classify further data. From the viewpoint of test oracles, these techniques have been applied to execution profile data to classify executions as different types such as “high”, “average” or “low” (for performance testing) [79] and “passed” or “failed” (for functional correctness) [79][80][81]. Many of these techniques focus on low-level execution profile data such as function calls, variable values and relations of these properties [79] [80]

[81]. Although some of these studies have shown promising results, they are hard to generalize, as the studies are limited with regard to properties such as the types of faults and types of programs considered [79] [80] [81]: mainly how the SUT data can be encoded in a suitable format and how useful the provided oracles are with respect to the specific features in the SUT specification. This type of oracles can thus be useful, but the user needs to consider their limitations.

2.1.4 Test harness

A test harness for a SUT has several roles including setting up the initial state of the SUT for each test and setting up the testing environment (SUT and interacting components). This section provides a brief overview of test generation related to test harness functionality from the viewpoint of the work presented in this thesis. This means that the focus is on setting up the collaborating components and their expected interactions (the SUT environment and related behaviour). Other viewpoints such as setting up the SUT state for a chosen unit test, as described in [82], are excluded from the scope of this thesis and thus not included in this overview.

One basic function of a test harness is to isolate the unit under test from the rest of the system. This is typically done by using a set of test components referred to as test stubs. When these are made programmable, they are often referred to as mock objects [83]. This means that a component library is used that provides interfaces to create these stubs and that the stubs can be programmed with expected interactions to simulate possible interactions between the SUT and its environment.

Tillmann and Schulte [84] use static analysis (symbolic execution) for automated generation of mock objects. By analysing the source code to see how it interacts with other objects, they infer the specification needed for the creation of the mock objects. By focusing on one test at a time, they also generate the expected behaviour of the mock object for that test, allowing the generation of mock objects to isolate the chosen parts for that test.

Saff et al. [85] use dynamic analysis to capture a trace of the SUT behaviour and use that as a basis to generate mock objects, their behaviour expectations and return values. Their goal is to factor larger tests into smaller tests in order to optimize the test suite execution and analysis. They use existing test cases as bases for analysis, capture traces of their execution including the passed objects

between the SUT and its environment, and use this information to define the expected behaviour of the mock objects and their return values.

Beyer et al. [86] generate test drivers to act as a test harness. Functions in the SUT interfaces and the interfaces of the environment with which it interacts are associated with data vectors. The external functions (of the environment) with which the SUT interacts are replaced with special test data feeding functions. Data from the test vectors are given in the order that the associated function is invoked. A similar approach is described by Pesonen [87] who uses instrumentation to isolate components from their environment, and captured data from actual use as test data.

Bertolino et al. [88] present a test harness generation method for service-oriented mobile applications. They require the system to be described using the web-service description languages WSDL and WS Agreement. Their goal is to facilitate testing of mobile systems where the environment is highly dynamic as a result of moving from one context to another. Based on these specifications, they generate test stubs for components with which the SUT needs to interact. They use simulators to further generate data to test the SUT in situations in which the test stub components are mobile and not always available.

2.1.5 Model-based testing

All testing can be considered to be based on models of the SUT. For example, Binder [89] describes testing as always being based on a model, even if it is only an implicit model in the mind of the tester describing the SUT and how it should be tested. Model-based testing is a technique that describes the SUT with the help of formal models at a higher abstraction level than the implementation and uses tools to analyse these models and generate tests from them [4]. Different definitions of MBT include generation of test data, invocation sequences, combinations of these including test oracles and turning abstract representations (such as UML diagrams) into test cases [4]. In the context of this thesis, the third definition (generation of data, sequences and including oracles) is used.

In this case, the model is an abstraction of the actual SUT, omitting excess details and describing only the relevant parts of interest for test generation. Using a set of coverage criteria and test-generation algorithms, the MBT generation tool generates test cases from this model. Coverage criteria may be, for example, covering all transitions in the model and algorithms including symbolic execution and graph traversal algorithms [4]. This generates a set of abstract test cases

that are then transformed into concrete test cases and mapped to the actual SUT by an adapter component.

MBT tools employ different types of models and test-generation techniques according to their intended application [90]. In the context of this thesis, the use of extended finite state machines (EFSM) is the most relevant one. A short description follows of these models and related MBT techniques according to [4]. EFSM's describe the SUT as a set of states, transitions between these states and the constraints defining when these transitions can be taken. The MBT tool then uses analysis algorithms to generate SUT invocation sequences, test data for these invocations and check the provided results. The used modelling notation is important as it needs to support all these properties, and programming languages such as Java are typically used to express them due to their ability to express all these properties [4][90]. In many ways, the MBT field brings together a number of previously presented research topics. Algorithms such as symbolic execution and constraint solving are used by MBT tools to analyse the model, generate tests and assess the coverage criteria [4]. It is also possible to combine these with various test data generation methods to generate input data for the generated test sequences.

Various representations are available for describing the models in MBT. Control flow is typically modelled with transition-based modelling languages such as state machines, and this notation is extended with a programming-language-like notation to model related data values [90]. This is a hybrid solution, and some tools also support the description of the complete model using a programming-language-like notation [4].

MBT has been an active research topic for a considerable time already. For example, Neto et al. [91] describe the work of Ramamoorthy et al. [92] in 1976 as an example of an early form of research into MBT. It has become more popular, however, especially in the last few years, and successes in its use have been reported in various domains including space [93], automotive [94] [95], healthcare [96] and others [91]. Despite several studies, however, industrial adoption is still seen to be lacking [91].

In MBT the model is typically created manually based on SUT specifications [4]. Creating an abstracted model of the SUT and using it to generate tests is seen as cheaper and less time consuming than writing similar tests manually. The acquisition of the required specialist skills for effective modelling, and creating and maintaining the models involve significant costs however. The opposite approach can also be taken where an initial model for the SUT is automati-

cally generated based on observing a set of controlled sample executions. This approach was presented by Bertolino et al. [5] as anti-model-based testing. They describe three reasons for taking this type of approach: the required deep expertise in formal methods for creating a model, the difficulty of forcing the actual SUT to take execution paths similar to those generated from the manually created model, and the lack of models for legacy systems and COTS-based systems [5]. In addition, this approach can be seen as having potential for significant cost savings, as the initial model is generated automatically. Besides describing the basic concept, Bertolino et al. do not take it further. This thesis presents a framework, which includes a practical tool implementation and evaluation of this type of testing.

2.2 Observation-based modelling

The building of models based on captured observations of a program is a popular approach in several fields of research such as test automation, program comprehension and reverse engineering. Reverse engineering, for example, is commonly defined as analysing a system in order to identify its components and their interrelationships, and creating representations of the system in another form or at a higher level of abstraction [97]. Similarly, program comprehension is a field that focuses on building a human understanding of a SW system. Storey characterizes it as theories to explain how programmers understand software and tools used to assist in these comprehension tasks [98]. The end result of these is typically a model that describes the SW under analysis at some abstraction level with the intention of helping the human user understand it better. This section is a review of how different techniques in different fields use observations captured from the SUT execution to build models of a program, and how these models are used for different purposes.

Figure 5 shows a generic overview of an observation-based modelling process. This process always starts with the definition of a set of execution scenarios to be used as a basis for the observations. These scenarios are used to drive the SUT execution and can include such elements as existing test cases or captured field data for the SUT [99]. During the execution of these execution scenarios, the SUT is monitored with a monitoring tool to capture the information required for the observation-based modelling technique. These tools can include basic

instrumentation frameworks such as AspectJ2, dedicated frameworks such as the one described in [100] or built-in support with the used middleware [21]. The captured observations are provided to the model-generator component or tool in a format suitable for automated processing. This produces, as a result, the target model, which can be used in different ways including testing and program comprehension. These results can also be used for improving or analysing the SUT, for example, by creating new test cases for [101] or optimizing the SUT behaviour [8].

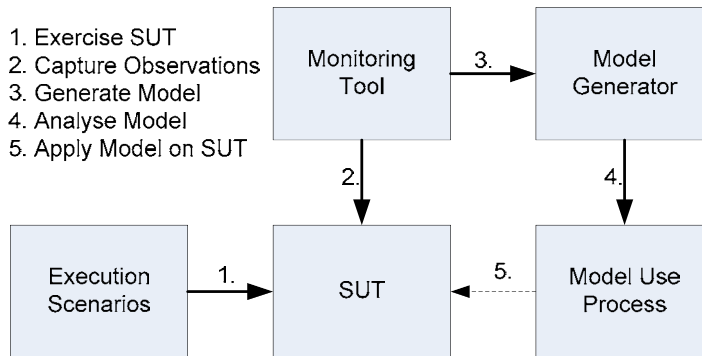


Figure 5.Observation-based modelling process.

Many of the techniques presented in the previous section build behavioural models of the system. The machine-learning techniques ([79], [80], [81]), for example, build models that classify a system into categories such as “pass” or “fail” in the context of software testing. More generally, many different fitness functions have been applied in a software engineering context, such as measuring the simplicity of the design (for understanding), and metrics such as cohesion and coupling as a basis for re-structuring a system [55]. Similarly, the previously described approaches of inferring invariants based on the execution traces can be used to create models of the system, describing the different data properties of the SUT and their relations (for example, [43], [66], [76]). As these modelling approaches have already been covered in the previous section, they are not repeated here.

As mentioned, the techniques for observation-based modelling produce models at different abstraction levels. The choice of abstraction level and the focus

[2] http://www.eclipse.org/aspectj/

of the analysis are important considerations for practical industrial adoption of any of these techniques. The focus on an overall view of a complete SW system requires a higher level of abstraction for the used models, as the use of detailed information for any non-trivial system will produce models that are difficult to comprehend for human users. If, however, the goal is to analyse smaller parts of a SW system for purposes such as debugging or unit testing, it is better to use more detailed information as a basis for the analysis. Thus, in order to scale techniques for observation-based modelling, the needs at different levels of analysis must be considered. This can be addressed with abstraction methods such as sampling, clustering and focusing on higher level information such as external vs internal interfaces. From the viewpoint of this thesis, different approaches have been applied in the different studies, such as detailed information for resource usage analysis [100] and higher level information for external interfaces [101]. This also relates to the intrusiveness of capturing the observations, as large-scale observation of low-level details can be very resource-intensive and disturb the system to the extent of making the results less useful. Depending on the goal, it is possible to change the abstraction level by, for example, focusing on class or component interfaces depending on the granularity of the analysis subject.

2.2.1 Basic terminology

Two closely related concepts that are often mentioned as supported in the tools and techniques reviewed in this section are concept assignment and feature location. SW program concepts can be mapped to two types: programming-oriented (such as searching or sorting a data structure) and human-oriented (such as reserving an airplane seat) concepts [102]. The problem of identifying human-oriented concepts and assigning them to their implementation in a program is termed as the *concept assignment* problem by Biggerstaff et al. [102].

A closely related term is *feature identification,* which aims to map program features to its source code [36]. A feature in this context is defined as a requirement that can be executed and observed [36] [103]. Techniques for feature identification start by defining a set of execution scenarios to represent the features in the system. These scenarios are executed, and a trace of the program execution collected for each scenario. A scenario is seen as corresponding to certain features of the program and the traces of executed code in different scenarios are thus used to map the features to their implementation. A basic method for feature identification is the *software reconnaissance* method presented by Wilde

and Scully [35]. In this method, the program is exercised with two types of execution scenarios: ones that exercise the feature under investigation and ones that do not. The differences in execution are used to focus the analysis on parts that are most likely to be a part of the feature under investigation. Other approaches include execution scenario trace difference analysis [35], concept analysis [36], epidemiological approach [103] and impact analysis [104]. User-assisted approaches include concept graphs [105].

2.2.2 State-based models

Mariani et al. [106] describe a technique to generate compatibility tests for COTS components based on the execution of previous versions of the components. They call this technique behaviour capture and test. Based on execution traces of the component, they generate IO and FSM models to describe the component behaviour. The IO models they use are the invariants provided by Daikon over the recorded data values that are changed during the component interactions. These include the values, recursively, of any passed objects. The FSM model generalizes all the recorded interaction sequences and is inferred from the trace with their kBehaviour algorithm. They use these models for regression testing of components by comparing models inferred from the previous and new versions of the component. This can be a new version of the same component or a new component from a different COTS provider. They use the identified violations as a basis to identify potential issues in component integration.

Lorenzoli et al. [107] present an algorithm called “GK-tail” that can be used to generate an EFSM from execution traces. This is based on finite state machines (FSM) and program invariants inferred with Daikon. Their algorithm combines the traces to form an FSM using a specific algorithm based on combining the sequences of method invocations of a given length to form the transitions of the FSM. Daikon-inferred invariants for passed and global data values are produced for each of the transitions in the FSM. These act as the EFSM constraints defining when a transition is allowed to take place. They consider using the EFSM for test case selection and for building an optimal test suite from existing test cases in order to increase the coverage of the model.

Xie and Notkin [108] present a model called Object State Machine (OSM). From a set of SUT test executions, they capture calls from the test cases to the SUT, parameter values and global state after each call. The global state is recorded in the form of capturing return values of all public methods with non-

void return type, including recursive scanning of composite objects. Method calls and their parameters represent state transitions and global state values from the states themselves. They see these models as useful for various tasks, such as finding failure causes by inspecting unexpected exception states and better understanding of the SUT.

Cook and Du [109] have demonstrated a technique to find points in a system that exhibit mutually exclusive or synchronized behaviour. They model the system as a state machine in which multiple states can be active at a time and each active state represents a concurrent execution path. The event trace used to build the model consists of the concurrent system states and transitions between the states. From these events, they use a technique to generate a state-based model for the system [110]. Once this model is established, they analyse the states to infer the points of mutual exclusion and synchronization. They intend that this model be used for facilitating the understanding of existing software systems.

Mesbah and Deursen [77] build an FSM of web-based user interfaces. They use a crawler tool to click through the interface and capture possible interaction sequences that cause changes in the DOM tree representation. A change in the DOM tree constitutes a new state, and the states of the DOM tree are also the states of the FSM. Transitions are the clicks (input) to the SUT that caused these changes in the DOM tree. They use this model as a basis for invariant-based test generation as described in the previous section.

Walkinshaw et al. [111] have presented a technique to infer state transitions from source code. The user must provide a set of states of interest, and their tool uses symbolic execution to analyse the source code in order to find transitions between these states and to describe the paths that lead to these transitions. States are identified using user-provided rules such as a certain method call triggering a state (transition). Walkinshaw et al. [111] consider the inferred models to be usable for various tasks, such as testing, documentation and program comprehension, but do not elaborate further.

2.2.3 Other models

Parsons et al. [112] discuss a number of techniques for producing an execution trace and capturing it. They also discuss turning these traces into models, including call graphs, runtime paths and calling context trees. The trace itself is described as the most detailed representation (model) of the SUT behaviour. A call-graph is described as a compact representation showing the SUT methods

and their calls to other methods. However, this is described as losing information (call sequences and context) available in the trace itself. Calling context trees is described as a compromise of these, preserving context information. They describe various uses for these models, including supporting optimization, reverse engineering, problem determination, autonomic management and redundant service removal.

UML has grown to be a popular notation for modelling software systems. It contains many different types of models that can also be used to describe SW behaviour. In the field of dynamic analysis-based modelling, two popular UML model types are sequence diagrams and state diagrams. State diagrams were covered in the previous subsection. Briand et al. [113] present a survey of tools to reverse engineer sequence diagrams, and Bennett et al. [114] present a survey and evaluation of tool features intended to help users in understanding reverse-engineered sequence diagrams. Briand et al. [113] describe the properties of the tools and their support in analysing the sequence diagrams such as support for viewing full control flow and execution pattern identification. They deem these to be important features, but due to various constraints such as the limitations of the UML sequence diagram notation find the support limited. Bennett et al. [114] note the lack of studies on the real support offered by different analysis tools for the user. For the tools, they list a number of goals to support the cognitive process of the user including design and architecture recovery, feature location, design pattern discovery and re-documentation at different levels of abstraction.

Most of the presented techniques analyse and model the behaviour of general properties of the SUT such as method calls and variable values. More domain-specific approaches have also been applied, with the basic strengths and weaknesses of domain-specific models. These have the benefit of providing a better fit for the chosen domain at the expense of excluding all other domains. One example of such an approach is analysis of the behaviour of telecommunication systems by Marburger and Westfechtel [115]. They use static and dynamic analysis with views such as dependency diagrams, link chains, state diagrams and sequence diagrams. Some of these are more general, such as sequence diagrams, and some are more domain specific, such as link chains. The dynamic analysis, in particular, focuses on domain-specific data such as signals and the data passed over the signals. They describe this using the models to aid understanding of the system, and found the data based on dynamic analysis especially useful.

Schmerl et al. [116] produce an architectural model from run-time events of a SUT. By mapping the set of run-time events to architectural events and trans-

forming these events into an architectural description, they produce the architectural model of the SUT. The architectural models they produce are described as sets of components and connectors.

Process mining is a technique aimed at discovering processes from event logs [117]. Although it originates from the field of (human-oriented) workflow monitoring, its applications have since been extended to various other domains, including SOA-based software systems [117]. This approach has been implemented in the ProM[3] tool that supports the building of various types of process models from given event logs such as Petri-nets and FSMs [118].

Lo et al. [119] mine temporal interaction rules from execution traces of programs. They call these invariants in describing statistically significant properties of interactions that hold over time in the program execution. These models consist of premises leading to consequences, that is, they describe how temporal event sequences (premises) are followed by other event sequences (consequences). They use the models to facilitate program comprehension, and as input for model checking to reveal errors in the implementation. Lo et al. [120] also use these temporal properties as input for the model-generation algorithms described in [107] in order to help improve the generalization of observed events from the traces and avoid the "spurious" events often observed in large-scale traces.

In addition to generating models from collected execution data of a program, another approach is to create these models separately and use them to support the analysis process. In this regard, the models are first created manually and then compared against an inferred model of execution. This can be used, for example, to document the current understanding of the system and to validate it against the actual execution. Koskinen et al. [121] have used what they call behaviour profiles to describe how classes are expected to interact in a system. They use UML models such as class and sequence diagrams to describe the profiles, and traces of program execution are mapped against these traces to see if they are correct. Counterexamples are also sought to find places where the behaviour is against that which is expected. The previously described approach by Roover et al. [75] uses a similar mapping of traces against predefined models, but uses logical queries written in Prolog to describe the model and for verifying these logic queries against the program trace.

[3] http://www.processmining.org

2.3 Discussion

This chapter focuses on describing the two main background concepts and the existing work on these concepts: test automation and observation-based modelling. These two fields share many properties, for example, as mentioned, Binder has noted that testing is always based on a model even if it is only an implicit one in the tester's mind [89]. In addition to this, the evaluation of a test execution is always based on observations. With reference to the previously mentioned definition of SW testing in the context of this thesis ("the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior." [7]), testing uses dynamic analysis as an underlying technique, that is, capturing observations from the execution of the SW behaviour and comparing these against set expectations. In this way, a form of observation-based modelling can be seen as always being required for the purposes of testing.

There is relatively little work on the type of observation-based modelling discussed in this thesis however. This means the approach of not writing the test expectations manually but using the captured observations from a set of execution scenarios as a basis to provide the expectations (the oracle information) itself. Some of the work on observation-based modelling that is described applies the produced model in the context of software testing (e.g. Lorenzoli et al. [107] for test suite optimization) or verification (e.g. Lo et al. [119] for model checking). While a few tools allow turning simple observation-based models into unit tests (e.g. [43]), to the knowledge of the author of this thesis, however, no previous work exists on providing (semi-)automated support for generating executable test models suitable for MBT from captured observations prior to the studies presented in this thesis. One of the biggest issues in this regard can be seen in the provision of useful new tests based on existing tests and other execution scenarios, as these already exist and are executable. In the work presented in this thesis, this issue is addressed through the combination of the different scenarios and the use of a model-based testing tool. The aspect of verifying the correctness of the produced models against the system specification (its expectations) is addressed with a set of guidelines forming a new method for applying the produced model for testing and verification of SW with the help of existing techniques related to observation-based modelling from the field of program comprehension.

As described, this thesis provides a construct as a combination of these two different fields. In this way, it provides a novel contribution to both of these fields in addition to the individual contributions of different publications.

3. A framework for observation-based modelling in model-based testing

This chapter presents an overview of the developed framework for observation-based modelling in model-based testing. More details can be found in the original research papers, as listed in the beginning of this thesis and provided as attachments. This chapter starts with a general overview and finally describes how the original papers contribute to the different parts of the presented framework. An implementation for EFSM models is used as an example while discussing the concepts at a general level. This implementation is made available as open source[4].

The process of applying this framework can be categorised into two distinct phases. In the first phase, the target model used for model-based testing is defined and tool support provided for the automation tasks. In the second phase, the available tool support and information on using the different concepts is applied. These two phases are described in more detail in the following subsections.

3.1 Phase 1: Defining the target model

The goal of observation-based modelling as presented in this thesis is to provide automated support for the modelling process in model-based testing. Different types of models can be applied in the test automation domain. Before applying model-based testing, it is important to define the target model that is being used and to provide automated tool support for generating the initial model.

In the first part of applying the process, the model and the required information for generating an initial version of it from captured observations needs to be

---

[4] http://sourceforge.net/projects/noen/

defined. This starts with choosing the model to represent the properties of interest for the SUT in order to enable testing of the required properties. This model is decomposed to define the information that needs to be captured as observations made about the system execution. This information is also mapped back to the target model in order to enable the building of suitable generation algorithms. To make this more manageable for complex models, they can first be decomposed into simpler models (behavioural patterns) that can be generated from captured observations and combined to form the target model. In the end, tool support is implemented to generate the initial model from the captured observations. This process is illustrated in Figure 6 and discussed in Paper VII.

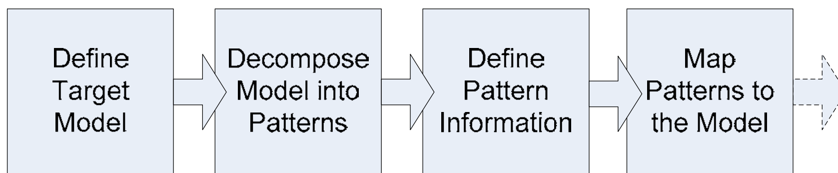


Figure 6. Model decomposition.

In this figure (Figure 6), the term pattern refers to behavioural patterns describing the different properties captured and inferred from running and observing the execution scenarios of the SUT. Here, a behavioural pattern refers to specific abstracted models that describe partial properties required for generating the complete target model. This includes describing the SUT interactions, data, their relations and similar properties. Each of these provides a set of different patterns that are later mapped together to produce the final target model. For example, using a set of captured input and output messages as observations, an FSM can be considered to be a behavioural pattern in which the different states and transitions describe the relations and behaviour of the SUT in terms of messages passed. Similarly, invariants over the data values processed by the SUT can be used as behavioural patterns describing the properties and relations for the data values. In this case, an FSM behavioural pattern can, for example, say that a request is always followed by a reply. A model in this context is considered to be more complex, for example, combining the properties of the behavioural patterns represented by an FSM and the properties represented by a data invariant model.

Once a target model has been decomposed and related tool support has been provided, this can be reused for different target systems. Once the information that needs to be captured has been defined and the tool support to generate the

models from this information has been implemented, it can be applied across different SUTs to generate the initial model for MBT. This thesis presents an implementation of one type of model. This model is an extended finite state machine (EFSM), which was described earlier in Section 2.1.5.

As described in Chapter 2, a test automation framework requires the existence of a number of different components such as test input data, a test harness and a test oracle. As was also described, the exact components and their requirements depend highly on the type of testing performed, the functionality under test, the design of the SUT and similar properties. For this reason, it is not possible to provide a generic set of components that would be applicable for all different testing purposes. This thesis, however, provides a basis for creating this mapping, provided the requirements for the model and the candidate models are known.

Table 3 shows the decomposition of the EFSM model into a set of behavioural patterns and the way they are mapped to a set of captured observations. Table 3 also briefly summarizes the basic parts of a test automation framework as presented in Chapter 2 and implemented in the EFSM model generator case example. These are combined to form the complete EFSM, which is implemented in an automated tool. The application of the model provided requires some special attention and is described in the next section, which describes the second phase of applying the OBM framework.

Table 3. EFSM model decomposition.

Model Element	Pattern	Observations
State	Data invariants	Data values representing the SUT internal state during each observed (input and output) message passed through the SUT external interfaces.
Transition	FSM	Input and output messages passed through the SUT external interfaces.
Transition guard	Data invariants	Input data values for received input messages, grouped as a separate invariant data point for each input-output message tuple.
Input data	Data invariants	Input data values (e.g. value ranges) used in input messages.
Test harness	Interface definitions	Messages defined in the SUT external input and output interfaces.
Test oracles	FSM and data invariants	Output messages (expected interactions) and their data values (expected return values). Associated separately for each separate transition.

3.2 Phase 2: Applying the framework

The toolset developed in the first phase is applied to the second phase. This phase can be repeated for any SUT to which the same type of model can be applied once the first phase has been completed. An overview of the process of using the toolset and the modelling approach is shown in Figure 7.

This second phase consists of three separate steps, each with a set of own substeps as shown in the figure. The arrows in the figure are labelled with a number for the step (1–3) and a letter for the substep inside the phase (a–e). Each arrow describes an activity for a step in the application of the framework. The boxes describe different entities related to the application of the framework, indicating the required tools, inputs and outputs that the steps use or produce.

This process of applying the framework is supported with automated tools for each step. The first step is about collecting the required data (observations) to build the model. This can be automated to different degrees depending on the type of observations that need to be captured. The second step generates the model. This step can be completely automated with tool support from the first phase. In the third step, the model is manually refined and executed (tested) until all errors found in the implementation and/or the specification are found and fixed. The refinement in this phase is manual, whereas the model execution, testing and reporting of results is performed by a test automation tool. Each of these steps is described in more detail next.

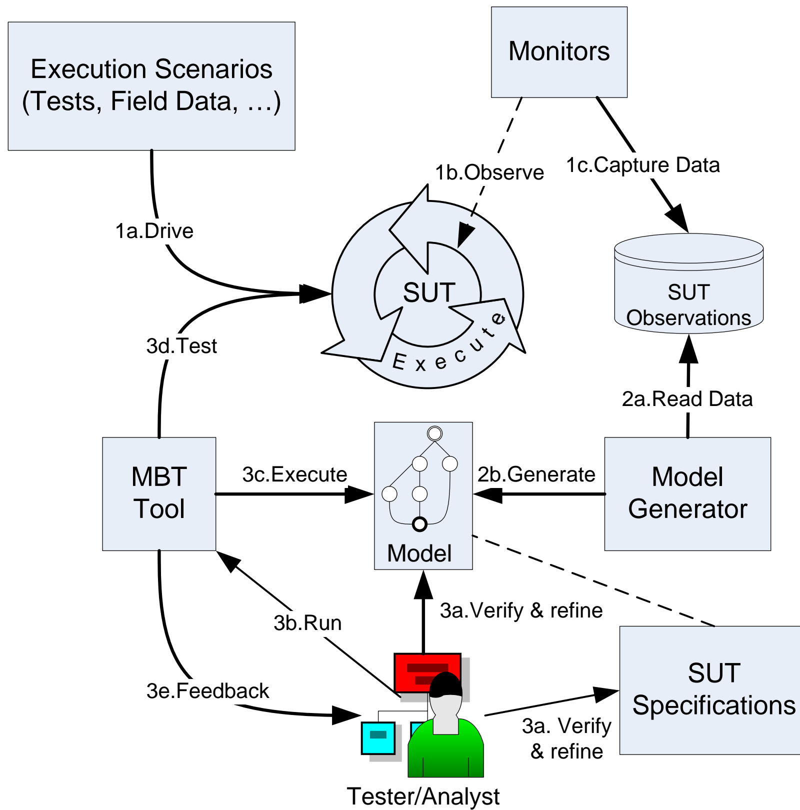


Figure 7. Overview of the framework.

3.2.1 Step 1: Capturing observations

The process of applying the framework begins in step 1 with the capturing of a comprehensive set of observations describing the behaviour of the SUT. In step *1a*, a set of execution scenarios are defined for capturing the base observations. Good candidates for these scenarios are the existing test suites of the SUT and field data captured, for example, from the use of the SUT [99]. Together these scenarios should form a representative set of the expected behaviour of the SUT from the perspective of the target model, including component interactions and input/output data. This means that the used execution scenarios should represent the complete behaviour of the SUT as it is intended to be included in the model, as opposed to only exercising a part of the SUT functionality. Depending on the requirements and the intended test target, the requirement for what constitutes a representative set can of course vary and is up to the expert defining the set, with

the help of tools such as the system specification. The completeness of these scenarios defines the quality of the model generated in the following step. The assessment of the set of observations has been described in more detail in Paper VIII. Guidelines for SW design to support more effective observations and testing are presented in Papers IV and V.

As the generated model is based on behavioural patterns mined from these observations, the mined patterns can be more powerful if the set of used scenarios provides a meaningful classification of expected results. For example, invalid input that causes error-handling behaviour can limit the usefulness of the generated model, as many existing tools for pattern mining do not provide functionality to mine sufficiently complex interactions between the input data values, interaction sequences and SUT internal state values. This means that unclassified scenarios will produce patterns and, as a result, a model that allows for all types of input and output, limiting their power of discovering failures such as errors for valid input. This is similar to existing work on using generated models for testing such as [66, 78] that use a set of observations considered to describe the correct behaviour of a SUT as a basis for the model, although different categorizations are also possible here. This is discussed in more detail in Papers VI and VII.

As in many cases, such classifications of available execution scenarios are not attainable; this is not a strict requirement. When the categorization is not available, it simply means that more manual work needs to be performed in the later phase of model refinement, for verification and testing to identify the categories in the produced model.

An execution driver component is needed to drive the execution scenarios of the SUT. This can be an existing test automation framework, a set of real users or any other form of available drivers. As the driver executes the SUT according to the execution scenarios, one or more monitoring components are used to capture a set of observations on the SUT behaviour (step 1b). As the framework is intended to take a black-box approach, it is sufficient to capture these observations from the SUT external interfaces and any provided test interface(s). The information captured at each observation point includes the internal state of the SUT, the input and output messages (method invocations) of the SUT and parameter values of these messages. Thus, it is most naturally applied to components that provide test interfaces to read their internal state, and that clearly define their external interfaces. Other options also exist, however, as described in Papers IV and VI. The observations for all execution profiles are stored in a data

store (step 1c). Steps 1a to 1c are (automatically) repeated by the execution driver component until all execution scenarios have been executed.

3.2.2 Step 2: Model generation

In the second step, the initial target model is generated based on the observations captured in step 1. In step *2a,* the model generator component uses the stored observation data as a basis to generate the initial target model in the notation used by the chosen MBT tool (step *2b*). In the implemented EFSM generator, this is the notation of the ModelJUnit[5] tool. The model generator mines the behavioural patterns into which the target model is decomposed from the observations, and combines these to produce the target model as defined in phase 1. In the EFSM case, the generator first mines the FSM and the Daikon data invariants from the observations and combines them into an EFSM model in the notation used by an MBT tool. The FSM uses the SUT external messages as a basis for both the states and transitions between these states. For example, it can define that from a Request state (message) it is possible to transition to a Reply state (message). The data invariants are inferred based on the parameter values of the messages and their relations to the state of the SUT at each point. These invariants describe the relations between the parameter and SUT internal state values that allow each transition to occur. For example, it may say that to subscribe with a given client name there must be connected clients, and the given client name must be in the list of connected clients in the SUT internal state. These data invariants and the FSM form the basic behavioural patterns that are then combined to form the final target EFSM model. Once the initial EFSM model is generated, it is provided to the user for the manual refinement and verification step (step 3).

The user of the generated model is typically a tester or analyst working with the software. In many cases, the use of an MBT tool requires an understanding of its special notations and concepts, some of which can be quite complex and unfamiliar to the user [91]. In the EFSM framework case study, the choice has been to use models presented in the Java programming language, which is familiar to many developers. Similar MBT tools also exist for other languages such as C# [122], and the EFSM generator could be modified to also provide models in these languages. The use of a familiar language has the advantage of making the

[5] http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

adoption of the MBT approach easier, allowing people to work with a familiar notion and familiar toolset in addition to the framework, which already generates a model with much of the required content automatically.

This forms the basis for the process of using the generated models. This has been described in more detail in Paper VI.

3.2.3 Step 3: Model refinement for verification and testing

The third step of refining the model is iterative. It starts with the user taking the generated model and choosing which part to focus on first (step 3a). These parts need to be enabled in the model (for example, with the transition guards in the EFSM), while keeping the rest of the model disabled. In practice this translates to disabling states in the state machine by setting their guards to always return false (in case of an EFSM), thus enabling the user to focus on the chosen parts of the model. With the target parts of the model enabled, the user can run the MBT tool (step 3b), which will attempt to execute the model (step 3c) against the SUT in order to test it (step 3d). The user will then proceed to enable the rest of the model one part at a time (for example, a transition at a time in the EFSM). The execution of the model with the MBT tool will give continuous feedback for the expanding model (with more parts enabled). In this step, the initial model also needs to be generalized to fully match the SUT specification, as it will be constrained to only describe the properties available in the used execution scenarios. The refined and generalized model is verified against the implementation by executing it with the MBT tool, which will report any mismatches between the two.

The execution of the model gives feedback to the user, highlighting where in the model the error was found. The user has to compare what the implementation does, what the model expects it to do and what the specification states that it should do. From this, the decision has to be made as to whether the error is in the implementation, the model or the specification. The original generated model should match the implementation and the refined one should match the specification. Problems such as misinterpretations of the specification or the modelling notation can also cause problems however. This part of using the refined model is referred to in this thesis as verification of the implementation vs the specification.

A second viewpoint in addition to the verification viewpoint is that of testing. As the model is continuously executed during the refinement phase, it can be executed at any time with the help of the MBT tool. This typically generates tests that produce complex interaction sequences and data as input for the system

under test. This can reveal additional errors in the implementation, where the implementation (or its part) is not "wrong" with regard to the specification, but where the different parts of the systems fail in the case of a more complex interaction sequence than has been tested before. In this thesis, this part is referred to as the testing part in the use of the model.

The method for using the generated model for software testing and verification has been described in more detail in Paper VI.

3.3 Discussion

As the generated model is based on observations made from the existing implementation, it also describes the actual implementation. Thus, care needs to be taken that the model is not taken as a specification of expected behaviour without consideration. If the implementation is not in accordance with the specification, this will not be visible simply by comparing the generated model with the implementation. The user must carefully verify that the model matches the specification, which should define what is expected of the implementation.

This approach makes use of model-based testing tools and techniques, but turns the basic MBT approach around. In MBT, the specification is normally used as a basis for the model, the generated tests are executed against the implementation and the results are verified to determine if any reported errors are due to problems in the model or in the actual implementation. As the techniques presented in this thesis turn this approach around, the model is now generated based on observations made from the actual implementation and, as described in this thesis, this model is refined to match the specification and constantly executed to see that the implementation still matches the specification. Thus, it is a change from matching the specification to the implementation to matching the implementation to the specification. This thesis addresses various aspects required to produce the observations, provides an implementation of the model generation, guidelines for use, and experiences and empirical evidence on their application.

Experiences from the executed case study indicate that the presented technique has potential to uncover different types of errors in both the implementation and the specification. In the case study, it effectively highlighted ambiguous parts of the specification that needed to be updated to define what was expected of the SUT. Several errors were found in the implementation where the specification was implemented incorrectly. Problems were also found in the way the SUT was designed and implemented in the more complex scenarios with gener-

ated input data and message sequences. These caused errors in the design at a level that was not clearly defined in the specification, but they were obviously not correct. Finally, some specified functionality was found to be missing completely from the implementation. This can only be found by checking that the required functionality is present in the generated model, as any unimplemented features will never be generated based on the implementation. These types of errors are best found when the SUT implementation and model generation/refinement are performed by different people, as it forces a new interpretation of the specification.

Table 4 summarizes the contributions of each paper to the framework.

Table 4. Contributions of each paper to the thesis subject.

Paper	Phase/ Step	Contribution
I	x	This paper presents a motivating case study from testing and integrating a complex real-world application. Many of the issues discussed are addressed in the later papers.
II	(2/1b), (2/2a), (2/2b)	This paper presents some of the initial concepts for the framework presented in this thesis that are taken further in later papers.
III	2/1a	This paper discusses the different properties of execution scenarios (test cases) and the way different types of scenarios are needed to cover different parts of the SUT effectively.
IV	2/1b	This paper presents guidelines for building support for effective monitoring into the SW at various levels from high-level architecture to the detailed design.
V	2/1b, 2/1c	This paper presents the design and implementation of a monitoring framework that provides services for generic system monitoring and supports the building of more specific monitoring.
V	2/2a	From the presented monitoring framework, it is also possible to export the observation data to different tools, including the formats used by the EFSM case study tools.
VI	2/1-3	This paper describes the overall process of phase 2, starting from the first steps up to the final steps, including a practical implementation for the EFSM case study.
VII	1	This paper describes the first phase of model decomposition using the EFSM model decomposition and generation as a case example while discussing the concept more generally.
VIII	2/1,3	This paper discusses means to analyse the set of observations, to optimize the set of used execution scenarios and to generate test oracles more generally.

4. Introduction to original papers

This chapter briefly summarizes the papers presented in this thesis and shows their contributions and relations to each other.

4.1 PAPER I: Integrating and Testing a System-Wide Feature in a Legacy System: An Experience Report

Paper I is an experience report on a project in which a feature with system-wide effects was integrated and tested on an embedded real-time software platform. The research format is that of a descriptive case study chosen to report the experiences considered interesting from a research perspective as encountered during the development work of the described project. The publication focuses on the problems and the reasons for them encountered during the course of this work. There is special focus on the problems caused by the environmental constraints of the software platform and the context in which the work was done. The highly embedded nature of the effected software coupled with its weak testability support made it difficult to control and observe the system. It was not possible to run any parts of the software effectively outside the embedded devices for which it was designed, nor was it possible to create different configurations of components to test the integration. The difficulties of decoupling the different software components from their execution platform and from each other also made all the real-time requirements of the platform unavoidable. Further problems were presented by the complex interactions between the system components. The wider context of the work included the integration of various black-box components from different vendors. There was very little visibility as to how the black-box components worked, and together they formed a network of components with complex interactions. This paper provides background motivation and requirements for the research area of this thesis. Similar issues were also reported in the interviews described in Paper IV. The author is the main

writer of the publication and was involved in the practical integration and testing work, and also collected and analysed the information presented in this paper from the practical project experiences. Mr. Mika Hongisto was the project manager and contributed hardware-related technical background descriptions for the paper. Mr. Kari Kolehmainen was the main person responsible for the actual integration and testing work described in the paper and provided discussions and insights into the topics described in the paper.

4.2 PAPER II: Towards Trace-Based Model Synthesis for Program Understanding and Test Automation

Paper II describes the concept of using traces of program execution as a basis for producing models of the executed program. The research approach in this paper is to design a construct to improve regression testing of a system, and it is a case study focusing on a single system. The main research motivation for the study came from the extensive, repetitive work of integrating and testing the project described in Paper I, which could have been aided greatly by a simple system such as that described here, considering the constraints that are also described in this paper (addressed in more detail in Papers IV & V). Validation is based on the application of the presented approach on regression testing of an actual failure in the case study system. The project chosen as a case study is different to the original industrial project, as this was no longer available to the researcher at this point. The basic idea is to capture information about the flow of the program based on a set of described instrumentation levels. The SUT is instrumented using any combination of instrumentation starting from means provided by the HW platform to manually inserted SW instrumentation. The captured data are used as a model of the way the system behaves, and this model is used as a basis for regression tests. This model is not yet suitable for use with existing MBT tools but rather requires a toolset of its own. It is best applicable for regression testing and to systems for which there is limited control over inputs and outputs. The basic concepts presented in this paper were further developed in later papers to be better suited to MBT. The author is the sole writer of the publication.

4.3 PAPER III: Towards a Deeper Understanding of Test Coverage

Paper III discusses how the commonly used basic measures of test coverage can benefit from a deeper analysis of the way different parts of the SUT are covered by the different tests. The research approach was that of using conceptual analysis to find properties relevant to the testing and test coverage of the SUT and to use this information to construct a design artefact for measuring test coverage at more detailed level(s). The motivation was to improve the measurement of test coverage and thus the analysis of where new tests are needed based on the author's experiences of various SW projects and in the SW testing community. Validation was based on analysis of the test suite and its coverage for an open-source project. The case study system was chosen for the availability of a suitable test suite and non-trivial complexity for providing a realistic environment. The roles of different tests such as unit tests, integration tests and system tests are discussed in order to show that it is not enough just to know that a part of the SUT is covered by some test, but that we need a deeper understanding of the types of tests in the test suite and how they cover the SUT together. The concepts of test granularity and level of testing are introduced to describe and compare the coverage of different tests and how they cover the SUT. The described measurements are implemented, and an open source project and its test suite are analysed as a case study of the proposed concepts. This paper provides the basic discussion on how we need different types of executions (tests) to cover the different functionalities and properties of a SUT when using the traces of its executions as a basis for building a model for its behaviour. Paper VI makes use of this information to select the types of executions that are used to build the model of the SUT. The author is the sole writer of the publication.

4.4 PAPER IV: A Study on Design for Testability in Component-Based Embedded Software

Paper IV is a study and a comparison of how design for testability (DFT) was addressed in two large-scale software companies working on embedded real-time software. The research approach is a survey focusing on exploring the practices of chosen companies in the field of interest. The choice of these companies was based on the industrial relations available at the time, and the choice of teams and projects inside these companies was based on their own expertise in

choosing a representative set with advanced solutions developed for the studied topics. The motivation for the study came from many sources including the author's experiences from several projects such as the one described in Paper I, and from the needs and interests of the industrial partners. The results are categorized into three different types: built-in functionality to support test implementation in the SUT, control of messaging between the system components with supporting middleware solutions, and simulation strategies that define how the software components can be integrated and tested outside their embedded device platform. Supporting test functionality ranges from libraries with readily built functionality to be integrated into the SUT as needed to first-class features in the SUT that will be part of the production code. These features provide functionality and interfaces to access information about and control SUT behaviour. Messaging solutions describe middleware solutions for the SUT that enable the composition of different combinations of the SW components and their test stubs and the control of the data and control flow as messages pass through the system. Simulation strategies start from the definition of the software platform in a way that allows the components developed for it to be executed without change in an external (desktop) environment. The solutions employed by the two companies are discussed and comparisons made of their effectiveness. Some of the information presented in this paper also served as the basis for the concepts developed and presented in Paper V. Some of the concepts presented are applied in Paper VI. The author is the sole writer of the publication.

4.5 PAPER V: A Probe Framework for Monitoring Embedded Real-Time Systems

Paper V presents a monitoring framework for embedded real-time systems. The research approach in this case focuses on improving issues described in Paper IV. Some of the motivation is also related to the issues presented in Paper I, where there was no effective support built into the system for monitoring and testing the system. This paper then focuses on effective support for the required features based on the previous papers. The case study subjects were chosen based on the industrial partners in the project at the time, which provided environments similar to the one presented in Paper I with mainly real-time embedded software. Validation was based on two analysis cases on actual systems provided by the industrial partners. The presented framework provides generic services that can be deployed as is to provide information about the system at a general level

and supporting services to implement any system-specific probes. It also supports integration with probes created with third-party tools such as SystemTap. SystemTap is used as an example as it enables a running SUT to be modified to add and remove probe code from the kernel. The described framework is implemented in C on a Linux platform and provided as shared libraries. The basic protocols are also provided as a Java implementation. These implementations are part of a larger framework, which enable the monitoring of a SUT, collection of the trace data and their exportation in different formats to different analysis and modelling tools. This also supports the Daikon and ProM tools used in Paper VI. Mr. Markku Pollari was responsible for implementing the presented framework on Linux and was the main writer of the publication. The author defined the concept of the framework to be developed, guided and assisted with the design and implementation, implemented the Java version and participated in writing the publication. This paper received a best paper award at the conference, and an invitation for an extended version by the Journal on Advances in Systems and Measurements.

4.6 PAPER VI: Observation Based Modeling for Model-Based Testing

Paper VI brings together many of the concepts presented in the previous papers. The research approach started with a conceptual analysis (study) of the MBT approaches in order to provide a basis for understanding the requirements for generating a model usable for MBT. A tool and a method for its use are designed, based on this information, and presented for the generation of both an initial model suitable for MBT and a method for using this model for testing and verification of SW behaviour. The choice of case study project was based on the industrial connections available at the Delft University of Technology, which the author was visiting at the time. Validation is based on a number of approaches, including actual testing of two SW components with the help of the provided artefacts, and simulation with the help of injected faults by mutation testing. The basic aim is to improve the support for MBT by making it easier to create the models and use them effectively for testing and verification. It presents a model-generation tool and technique for generating an EFSM for MBT based on the observations captured from the SUT execution. It is an extension of the work described in Paper II of using traces of the program execution as a basis for building models for its behaviour. Different types of execution scenarios, as

described in Paper III, are used as a basis to provide the observations to build the model. DFT solutions, as described in Paper IV, were used to capture the information required for the trace used as a basis for the model generation. Although the case study is executed on a different platform, the framework presented in Paper V supports the capture of the required observations and provides them in a suitable format for the model-generation tools applied. The presented technique involves the user instrumenting the SUT, capturing a set of observations based on a set of execution scenarios, using the provided tool to generate an EFSM from these observations and refining the EFSM manually to its final representation. The SUT specification is used to verify the correctness of the generated model (and thus of the implementation from which it is generated) and an MBT tool is used to generate more tests to further explore the generated EFSM. Using a case study, it shows how the technique can effectively find errors in both the implementation and the specification. The author is the main writer of the publication and defined the concepts, designed and implemented the used algorithms and tools, and performed the case study. Dr. Eric Piel provided the case study subject, helped execute the case study and participated in writing the publication. Dr. Hans-Gerhard Gross also helped to provide the case study and participated in writing the publication.

4.7 PAPER VII: Behavior Pattern-Based Model Generation for Model-Based Testing

Paper VII is both a generalization and a deeper description of the model generation technique presented in Paper VI. The research approach of this study starts with the use of conceptual analysis to identify the relevant properties of test automation systems and models in order to provide a basis for the provision of a generic framework for developing algorithms to generate models for model-based testing based on captured observations. A generic framework is designed, based on this information, and presented in order to support the creation of algorithms to support different types of models suitable for MBT. The choice of the case study subject is the same as in Paper VI due to the work being performed in the context of the same project. Validation is based on the application presented in more detail in Paper VI and on the description (via conceptual analysis) of its possible application to a second project related to web-application testing at the Delft University of Technology, which the author was visiting at the time of performing this study. While Paper VI focuses on describing the implementation

of the model generator and its application to the case study, this paper focuses on the process of generating different types of models. A general description of this process is presented from the viewpoint of generating different models for use with MBT, and the EFSM model generation is used as a case example for all the parts of the process. This process is described as decomposing a target model (such as EFSM) into a set of behavioural patterns that can be mined from the observations captured from running the execution scenarios for the SUT. For this, the information to be observed and captured as well as the means and algorithms for mapping this back to the target model (generating the model) are described. The process of using these generated models is also shorortly described and compared with traditional MBT approaches. The author is the sole writer of the publication.

4.8 PAPER VIII: Program Comprehension for User-Assisted Test Oracle Generation

Paper VIII describes synergies in the fields of program comprehension and test oracle automation. The research approach in this case starts from conceptual analysis of both program comprehension and test oracles, leading to the identification of relevant parts and properties of these two fields and their relations to each other. This is then used as a basis to design a framework for creating (semi)automated support for test oracle generation with the help of program comprehension tools and techniques. The motivation for this study comes from the observation that test oracle automation is one of the most difficult and least supported parts of test automation research. From the research work performed by the author to this point, it was clear that this part could be separated and provided individually as a meaningful and useful contribution to the field of test oracle automation. The choice of case study is the same as in Papers VI and VII as they are part of the same research work. The validation is also similar to Paper VII. While Paper VI focuses on a single type of model, this paper provides insights into taking this into a wider context. The paper focuses on describing the concept of providing automated assistance for the generation of test oracles by means of program comprehension tools and techniques. The tool and technique presented in Papers VI and VII are used as an example of how a model generated from a set of captured observations can be used as an aid to understanding the system and to turn this model into an automated test oracle. It also shows how the set of used execution scenarios and the resulting set of captured

observations can be optimized to provide a sufficiently complete set for generating the target model. The concept is also viewed against other related work in the literature, and a generalization of the approach is presented by relating the provided example to the other approaches and against the theoretical backgrounds of both test oracle automation and program comprehension. The author is the sole writer of the publication.

5. Framework evaluation

Different parts of the framework presented in this thesis have been evaluated through different study subjects. This chapter presents the different study subjects and the way they have been used in the evaluation of the different phases and steps of the proposed framework.

5.1 Study subjects

A number of different study subjects have been used during the evaluation of the different parts of the framework presented in this thesis. This section gives a brief overview of each of the study subjects related to each of the attached papers in the form of Table 5. As discussed in the presented overviews of the different papers in Chapter 4, Paper I also contributes to the evaluation of the different approaches in the papers by providing a set of practical problems from a real project with significant complexity and problems in integration and testing. A mapping of the different papers to these problem areas was also discussed in Chapter 4.

Table 5. Study subjects in the different papers.

Paper	Study subject
I	A SW platform for mobile devices is studied in this paper, including the platform code and a number of applications built on top of it. The process of integrating and testing a feature with system-wide effects was described, and experiences from this process were described to highlight problem areas that could be addressed in future research.
II	A middleware messaging component was studied. The use of captured traces (observations) and how they could be turned into regression tests for the SUT was explored. A case study with an actual error scenario for the SUT was used to evaluate the concept in practice.
III	An open-source software project, its existing test suite and test coverage were studied. The paper studied means for more detailed analysis of test coverage for a test suite.
IV	Several experts from two different industrial companies were interviewed to gather the best practices relating to design for testability. These experts had worked on a number of different projects and were chosen by the companies as the best representatives to provide information on this topic. In addition, technical documentation describing the implementations of these solutions was reviewed.
V	Experiments were performed on two Linux-based platforms. One was a desktop Linux environment and the other was an embedded Linux system provided by one of the industrial project partners. The use of the presented probe framework for monitoring the targeted systems, including the overhead caused, was studied both at the kernel and user-space level.
VI	The application of the presented model-generation approach and the proposed OBM method were studied with the help of different components providing services for a maritime surveillance system. This included studying the effectiveness and usefulness of the proposed approaches through the capability of finding previously undiscovered errors in the implementation and ambiguities in the specification. In addition, effectiveness in test coverage and mutant detection were evaluated in comparison with the existing tests used as a basis for model generation.
VII	A database and server component that is a part of a larger system is studied. This study subject is very similar to the ones used in Paper VI. This paper discusses the model decomposition and algorithm development aspects with the help of the case study in more detail and more generally. Paper VI provides a more detailed evaluation of this approach with a specific case study. In addition, this paper (VII) briefly describes the application of the presented concept in the domain of web-applications. This description related to web applications is based on ongoing work but is not discussed in more detail as it is work in progress.
VIII	A database and server component that is a part of a larger system is studied. This study subject is very similar to the ones used in Papers VI and VII. Again, Paper VI provides a more detailed evaluation of this approach with a specific case study. This paper (VIII) and its case study show how program comprehension techniques can be used to assist in semi-automated generation of test oracles more generally.

5.2 Phase 1: Defining the target model

Three studies presented in Papers II, VI and VII are relevant to the evaluation of the first phase. A simple and straightforward approach is presented in Paper II, which takes the interesting properties of the behaviour as captured in the trace (observations) and simply turns these directly into a regression test for the SUT. This is evaluated by applying it to the actual study subject and testing its error scenario.

A generic approach to more complex modelling is presented in Paper VII, which also discusses its application to the definition of the required information for generating EFSM models and for generating similar models for automated GUI-based testing of web applications. This is evaluated through practical application to these types of models, as described in Paper VII. A detailed evaluation of this approach is given in Paper VI, in which details of the different parts of the model decomposition, generation and application are given, and its usefulness is evaluated with the study subjects used.

Together, these studies provide both the generic guidelines for defining the target model and for decomposing it into the information needed, as well as evaluating it with the use of practical case studies.

5.3 Phase 2: Applying the framework

Different studies from the different papers are relevant to the evaluation of the steps of this second phase. The following subsections describe the evaluation viewpoint for each of these papers and steps.

5.3.1 Step 1: Capturing observations

Papers II, III, IV, V and VI are relevant to the evaluation of this step. Paper II discusses different means of capturing a trace (observations) from low-level hardware support to high-level application-specific functionality. In this case, application-specific functionality is used to build a basis for test information. Paper III discusses the different types of execution scenarios used to build the set of observations. A practical evaluation of different execution scenarios is presented for the analysed software, showing how this can be applied in practice.

Paper IV presents a set of guidelines for building support for testing and analysis into the architecture and design of a software system. This supports the

observations phase by allowing more control over the targeted system (focused observations, inputs and outputs) and building more advanced options and advanced support for the observation process itself. The evaluation is based on the discussions and experiences of the experts from the surveyed companies and the analysis of the provided information. Paper V is a continuation of this work, building a sophisticated framework that provides support to design these types of features into the different parts of the system being analysed. This (probe) framework is evaluated with two different case studies, showing how it can be used to provide this kind of support in practice and how effective it is. The results showed significant improvements over existing systems.

Together, these studies provide support for the observation-capturing step in a manner in which each part of the proposed approach(es) has/have been evaluated with empirical means.

5.3.2 Step 2: Model generation

Papers II, VI and VII are relevant to the evaluation of this step. As discussed, Paper II presents an early concept of a simple turning of a captured trace (observations) into a test model and using this as a basis for regression testing. It is evaluated with a case study, showing how the given approach can help in the discovery of actual faults introduced over SW evolution.

Paper VII provides generic guidelines for turning the captured information (as defined in phase 1) into a suitable test model for the SUT. This is evaluated with the provided case studies, showing how an EFSM can be decomposed into a set of behavioural patterns, which are combined to form the complete target model from the captured observations. Paper VI also provides a detailed evaluation of EFSM model generation with detailed case studies showing how this can be done, as well as evaluating its accuracy in producing a complete model suitable for SW testing and verification.

Together, these studies provide the generic guidelines for building the algorithms to generate the target model from the captured observations and for evaluating these with the use of practical case studies.

5.3.3 Step 3: Model refinement for verification and testing

Papers II, VI and VIII are relevant to the evaluation of this step. Paper II discusses some initial ideas of using the provided model to also support the process

of understanding the behaviour of the SW together with the goal of using it for testing the SW. This provides a basis for the concept of using a generated model for both of these purposes, which is important for their application in both SW testing and verification. This is evaluated with the analysis of a captured trace (observations) for the used case study, showing how the model can be used to support understanding of the analysed functionality and detect errors in regression testing.

Paper VIII provides a generic overview of the similarities of both the program comprehension research field and the test automation research field. It provides a comprehensive overview of both fields and their similarities based on existing work, which in itself serves to evaluate the concept. This is further illustrated with the help of a case study on analysing a generated EFSM model with the help of tools intended to support human analysis (program comprehension) and providing the (human) user with an option to turn these (machine-) generated models into complete test models to be used for model-based testing. This is evaluated in more detail in Paper VI, which provides a detailed study of using a generated EFSM model as a basis to verify the implementation of the SUT against its specification and to generate more tests to detect errors in the implementation.

Together, these studies provide both the generic guidelines for using the generated model as a basis for software testing and verification, and for evaluating these with the use of practical case studies.

6. Conclusions

This thesis presented a framework for observation-based modelling in model-based testing. It discussed automated generation of an initial model for MBT of SW and its application in testing and verifying the SUT. The introductory part presented the research framework and questions. The literature related to the research problems was also reviewed. A motivating case example of the problems of test automation for modern systems is presented in Paper I. The later papers then focused on different parts of the research questions, each with a different focus. The last few papers pull all the topics in the subquestions together to give an answer to the main research question. The main research question concerned the provision of automated support for model creation for model-based testing.

6.1 Answers to the research questions

The first subquestion asked how the required information to generate the models can be captured. The answer to this question is two-fold. As the focus is on generating models for existing SW systems, the focus is on analysing the implementation of these systems. First, Paper VII presented a decomposition of the target model to the required observations. For capturing the actual observations, the chosen approach uses dynamic analysis of SUT executions, allowing for a black-box, component-based approach. Paper III discusses the different types of execution scenarios needed to capture a representative model of a system. Paper IV discusses implementation and design solutions for capturing the required information, and Paper V presents an implementation of a framework that provides supporting functionality for these solutions. Paper VIII discusses the analysis and optimization of the set of execution scenarios and observations.

The second subquestion concerned turning this information into a model suitable for MBT. This question is addressed in Papers II, VI, VII and VIII. Paper II presents a straightforward approach of using a trace captured from the execution of the SUT as a flow-based model for regression testing. Paper VI presents a more sophisticated approach to generate an EFSM model from the captured observations of the SUT. A toolset to produce the required observations and automatically generate the initial model from them is also presented. The produced EFSM includes all the required elements, including a test harness, test input and test oracles. When the MBT tool is run, it generates test scripts with all these elements and executes them against the SUT. The model is generated in a form usable as such for an MBT tool. Paper VI thus focuses on the EFSM case study, and Papers VII and VIII on generalizing the different parts of this approach.

The final research question asked how the generated initial models can be used for SUT verification and testing. As the models are generated based on information captured from the execution of the SUT, they correctly describe its actual behaviour. This does not necessarily match the expected behaviour of the SUT however. The most likely source for the correct, expected behaviour is the specification of the SUT. The process of verifying this correctness is described in Paper VI and the approach used is described more generally in Paper VIII.

Together, the answers to these subquestions form an answer to the main research question. They present a complete framework for observation-based modelling in model-based testing. The answer to the first question describes how to capture the required information for the initial model generation. The answer to the second question describes how this information can be turned into a model usable for MBT. The answer to the third question shows how this model can be applied to software testing in a reliable way, allowing the user to verify the correctness and completeness of both the implementation and the specification. Together they answer the main research question by providing automated support for model creation for model-based testing.

6.2 Limitations and future work

The term MBT has many different definitions depending on who uses it and in what context [4]. Even with the definition used in this thesis, different approaches can be taken and different types of models can be applied [90]. An implementation for EFSM models was presented, with generic analysis and discussions for different types of models. Thus, guidelines for different types of

models are provided, but the implementations for these different types of models and different types of target systems are left for future work.

The creation of the observations (traces) for model generation requires a set of SUT executions that thoroughly exercise the behaviour of the SUT. Different properties for these executions and their sources are discussed in Papers III and VI. A good source is an existing test suite with categorizations of tests related to error handling and correct behaviour tests. Another option is to use data captured from monitoring the SW in its actual environment. In many cases, however, the availability of a good set of suitable execution data is limited. The test data generation methods presented in the literature review part of this thesis could be used to provide a basis for automatically generating a suitable set of executions. To be effective, however, this would require means to provide automated assistance for the classification of the produced inputs and resulting outputs to enable the generation of powerful models as discussed before. Tools and techniques, such as the machine learning and classification techniques described in Chapter 2, are one option to consider for providing these classifications. This area of research has already been discussed in Paper VIII. As such, it provides an interesting venue for future research.

The tools for mining the behavioural patterns used in the EFSM case study are intended to be generic and as such are not designed for the purpose of generating models for model-based testing. This results in limitations on their applicability and on the completeness of the models provided as well as a requirement for more manual refinement when using these models for SUT verification and testing. The specific limitations for the EFSM case study are discussed in Papers VI, VII and VIII, together with possible means to make the behavioural pattern mining more powerful in this case. In general, it can be said that making more specific behavioural pattern-mining tools for the purposes of using the patterns to generate specific models for model-based testing would make the automated model generation more powerful and provide more complete initial models. This also involves the trade-off of reusing existing tools (as done in the EFSM case study) and writing new specific tools for this purpose. Addressing this requires experiments with implementation and trying such pattern-mining tools for chosen models and systems.

One important property to study is more thorough evaluation of the gains of using the framework presented in this thesis. This involves performing user-based studies on the usefulness of the generated initial models vs writing the models from scratch, including the costs of acquiring the required skills for the

modelling. Possible research approaches for this include introducing OBM into the modelling process (action research) or a comparative case study with one group using the OBM approach and another a “traditional” modelling approach. Another interesting topic of study in this regard is that of different types of test coverage and of how much more of the existing system can be covered by refining the initial model and using a model-based testing tool to generate further tests from this model in addition to the initial set of execution scenarios used to capture the observations for the model generation. This has already been evaluated in Paper VI, but more experiments with different types of systems and parameters of experiments would be of interest.

When problems are found in the execution of the model, that is in the comparison (execution) of the model vs the implementation, both the model and the implementation need to be analysed to find out which one is incorrect. A technique for this is presented in Paper VI in the form of the creation of a separate test case that makes the inputs and outputs explicit as well as the message sequences used in the test case. By analysing and modifying this simple test case, it is possible to pinpoint the actual cause of failure more effectively. There is currently no automated support for this, but the separate test case needs to be manually created. As all the information required to generate this test case automatically is available in the test generated by the MBT tool, this process could also be automated to generate the initial test for debugging. This would be more in line with the off-line approach to MBT as the provided toolset is currently only used as an on-line testing tool.

Although the implemented approach is successfully tested on a real implementation, as discussed in Paper VI, this is only one tested component from a relatively simple research prototype system. More experiments on real systems of significant complexity are likely to reveal more constraints in the proposed techniques. Performing these experiments to further validate the work and addressing any constraints that are found would be a topic for future study.

References

[1] Bertolino, A. Software Testing Research: Achievements, Challenges, Dreams. In: Proceedings of Future of Software Engineering (FOSE07), 2007. Pp. 85–103.

[2] Knuth, D. Knuth: Frequently Asked Questions. [Online]. HYPERLINK: http://www-cs-faculty.stanford.edu/~knuth/faq.html.

[3] Gelperin, D. and Hetzel, B. The Growth of Software Testing. Communications of the ACM, Vol. 31, No. 6, pp. 687–695, June 1988.

[4] Utting, M. and Legeard, B. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2007.

[5] Bertolino, A., Polini, A., Inverardi, P. and Muccini, H. Towards Anti-Model-Based-Testing. In: Fast Abstracts in International Conference on Dependable Systems and Networks (DSN'04), Florence, 2004.

[6] Sommerville, I. Software Engineering, 8th ed. Addison Wesley, 2006.

[7] Bertolino, A. and Marchetti, E. Software Testing. In: Guide to Software Engineering Body of Knowledge (SWEBOK). Bourque, P. and Dupuis, R. (eds.). IEEE Computer Society, 2004, ch. 5. Pp. 5-1–5-16.

[8] Tuuttila, P. and Kanstrén, T. Experiences in using principal component analysis for testing and analyzing complex system behavior. In: 21st International Conference on Software & Systems Engineering and their Applications, Paris, France, 2008.

[9] Kanstrén, T. Towards a Deeper Understanding of Test Coverage. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 20, No. 1, pp. 59–76, 2008.

[10] Glass, R. L., Vessey, I. and Ramesh, V. Research in Software Engineering: An Analysis of the Literature. Information and Software Technology, Vol. 44, pp. 491–506, 2002.

[11] Runeson, P. and Höst, M. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, Vol. 14, No. 2, pp. 131–164, April 2009.

[12] Järvinen, P. On Research Methods. Tampere, Finland: Tampereen yliopistopaino Oy, 2004.

[13] Nunamaker, J. F., Chem, M. and Purdin, T. D. M. Systems development in information systems research. Journal of Management Information Systems, Vol. 7, No. 3, pp. 89–106, 1991.

[14] March, S. T. and. Smith, G. F. Design and Natural Science Research on Information Technology. Decision Support Systems, Vol. 15, pp. 251–266, 1995.

[15] Hevner, A. R., March, S. T., Park, J. and Ram, S. Design Science in Information Systems Research. Management Information Systems Quarterly, Vol. 28, No. 1, 2004.

[16] Ramesh, V., Glass, R. L. and Vessey, I. Research in Computer Science: An Empirical Study. Journal of Systems and Software, Vol. 70, pp. 165–176, 2004.

[17] Bealey, M. Analysis. In: Stanford Encyclopedia of Philosophy, 2009. [Online]. HYPERLINK: http://plato.stanford.edu/entries/analysis/ http://plato.stanford.edu/entries/analysis/.

[18] Kitchenham, B., Pickard, L. and Pfleeger, S. L. Case Studies for Method and Tool Evaluation. IEEE Software, Vol. 12, No. 4, pp. 52–62, July 1995.

[19] Harjumaa, L. Improving the software inspection process with patterns. Oulu, Finland: University of Oulu, 2005.

[20] Berner, S., Weber, R. and Keller, R. K. Observations and Lessons Learned from Automated Testing. In: Proceedings of the 27th International Conference on Software Engineering (ICSE05), St. Louis, Missouri, USA, 2005. Pp. 571–579.

[21] Kanstrén, T. A Study on Design for Testability in Component-Based Embedded Software. In: Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications, Prague, Czech Republic, 2008. Pp. 31–38.

[22] Binder, R. V. Design for Testability in Object-Oriented Systems. Communications of the ACM, Vol. 37, No. 9, pp. 87–101, September 1994.

[23] Persson, C. and Yilmaztürk, N. Establishment of Automated Regression Testing at ABB: Industrial Experience Report on 'Avoiding the Pitfalls'. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE04), 2004. Pp. 112–121.

[24] Bass, L., Clements, P. and Kazman, R. Software Architecture in Practice, 2nd ed. Addison-Wesley, 2003.

[25] Briand, L. C., Labiche, Y. and Sówka, M. M. Automated, Contract-Based User Testing of Commercial-Off-The-Shelf Components. In: Proceedings of the 28th International Conference on Software Engineering (ICSE06), Shanghai, China, 2006. Pp. 92–101.

[26] Rehman, M. J., Jabeen, F., Bertolino, A. and Polini, A. Testing Software Components for Integration: A Survey of Issues and Techniques. Journal of Software Testing, Verification and Reliability, Vol. 17, pp. 95–133, 2007.

[27] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming, 2007.

[28] Andrews, D., Briand, L.C. and Labiche, Y. Is Mutation an Appropriate Tool for Testing Experiments? In: Proc. 27th International Conference on Software Engineering (ICSE'05), 2005. Pp. 402–411.

[29] Do, H., Elbaum, S. and Rothermel, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering, Vol. 10, No. 4, pp. 405–435, 2005.

[30] DeMillo, R. A., Lipton, R. J. and Sayward, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer, Vol. 11, No. 4, 1978.

[31] DeMillo, R. A. and Offutt, A. J. Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, Vol. 17, No. 9, September 1991.

[32] Liu, M.-H., Gao, Y.-F., Shan, J.-H., Liu, J.-H., Zhang, L. and Sun, J.-S. An Approach to Test Data Generation for Killing Multiple Mutants. In: Proc. 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006.

[33] Santelices, R., Jones, J. A., Yu, Y. and Harrold, M. J. Lightweight Fault-Localization Using Multiple Coverage Types. In: International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009.

[34] Rothermel, G., Elbaum, S., Malishevsky, A. G., Kallakuri, P. and Qiu, X. On Test Suite Composition and Cost-Effective Regression Testing. ACM Transactions on Software Engineering, Vol. 13, No. 3, pp. 277–331, 2004.

[35] Wilde, N. and Scully, M. C. Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance: Research and Practice, Vol. 7, No. 1, January/February 1995.

[36] Eisenbarth, T., Koschke, R. and Simon, D. Locating Features in Source Code. IEEE Transactions on Software Engineering, Vol. 29, No. 3, pp. 210–214, March 2003.

[37] Zeller, A. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, 2005.

[38] Andrews, J. H., Briand, L. C., Labiche, Y. and Namin, A. S. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Transactions on Software Engineering, Vol. 32, No. 8, pp. 608–624, August 2006.

[39] Harder, M., Mellen, J. and Ernst, M. D. Improving Test Suites via Operational Abstraction. In: International Conference on Software Engineering, Portland, Oregon, 2003. Pp. 60–71.

[40] Duran, J. and Ntafos, S. An Evaluation of Random Testing. IEEE Transactions on Software Engineering, Vol. 10, No. 4, pp. 438–444, 1984.

[41] Gutjahr, W. Partition Testing Versus Random Testing: The Influence of Uncertainty. IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 661–674, 1999.

[42] Ciupa, I., Leitner, A., Oriol, M. and Meyer, B. ARTOO: Adaptive Random Testing for Object-Oriented Software, 2008, pp. 71–80.

[43] Boshernitsan, M., Doong, R. and Savoia, A. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In: Proc. International Symposium on Software Testing and Analysis (ISSTA'06), Portland, Maine, 2006. Pp. 169–179.

[44] Beer, A. and Mohacsi, S. Efficient Test Data Generation for Variables with Complex Dependencies. In: Proceedings of the 1st International Conference on Software Testing, Verification and Validation, Lillehammer, Norway, 2008. Pp. 3–11.

[45] Clarke, L. A. A System to Generate Test Data and Symbolically Execute Programs. IEEE Transactions on Software Engineering, Vol. 2, No. 3, pp. 215–222, September 1976.

[46] King, J. C. Symbolic Execution and Program Testing. Communications of the ACM, Vol. 19, No. 7, pp. 385–394, July 1976.

[47] Gotlieb, A., Botella, B. and Rueher, M. Automatic Test Data Generation using Constraint Solving Techniques. In: Proc. International Symposium on Software Testing and Analysis (ISSTA'98), 1998. Pp. 53–62.

[48] Tillmann, N. and Schulte, W. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. IEEE Software, Vol. 23, No. 4, pp. 38–47, July/August 2006.

[49] Korel, B. and Al-Yami, A. M. Assertion-Oriented Automated Test Data Generation. In: Proc. 18th International Conference on Software Engineering (ICSE'96), 1996. Pp. 71–80.

[50] McMinn, P. Search-Based Software Test Data Generation: A Survey. Journal of Software Testing, Verification and Reliability, Vol. 14, pp. 105–156, 2004.

[51] Korel, B. Automated Software Test Data Generation. IEEE Transactions on Software Engineering, Vol. 16, No. 8, pp. 870–879, 1990.

[52] Grindal, M., Offutt, J. and Andler, S. F. Combination Testing Strategies: A Survey. Journal of Software Testing, Verification and Reliability, Vol. 15, pp. 167–199, 2005.

[53] Ostrand, T. J. and Balcer, M. J. The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM, Vol. 31, No. 6, pp. 676–686, June 1988.

[54] Thummalapenta, S., Xie, T., Tillmann, N., Halleux, J. de and Schulte, W. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In: Proc. 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering, Amsterdam, The Netherlands, 2009.

[55] Harman, M. The Current State and Future of Search Based Software Engineering. In: Proc. Future of Software Engineering (FOSE'07), 2007.

[56] Buehler, O. and Wegener, J. Evolutionary Functional Testing of an Automated Parking System. In: Proc. International Conference on Computer, Communications and Control Technologies and the 9th International Conference on Information Systems Analysis and Synthesis, Orlando, Florida, 2003.

[57] Harman, M., Lakhotia, K. and McMinn, P. A Multi-Objective Approach to Search-Based Test Data Generation. In: Proc. 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), 2007. Pp. 1098–1105.

[58] Tonella, P. Evolutionary Testing of Classes. In: Proc. International Symposium of Software Testing and Analysis (ISSTA'04), Boston, Massachusetts, 2004. Pp. 119–128.

[59] Ayari, K., Bouktif, S. and Antoniol, G. Automatic Mutation Test Input Data Generation via Ant Colony. In: Proc. 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), 2007. Pp. 1074–1081.

[60] Baudry, B., Fleurey, F., Jézéqual, J.-M. and Traon, Y. L. From Genetic to Bacteriological Algorithms for Mutation-Based Testing. Journal of Software Testing, Verification and Reliability, Vol. 15, pp. 73–96, 2005.

[61] Bertolino, A., Gao, J., Marchetti, E. and Polini, A. Automatic Test Data Generation for XML Schema-Based Partitioning Testing. In: Proc. 2nd International Workshop on Automation of Software Test (AST'07), 2007.

[62] Sneed, H. and Huang, S. The Design and Use of WSDL-Test: A Tool for Testing Web Services. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 19, pp. 297–314, 2007.

[63] Yaun, X. and Memon, A. M. Using GUI Run-Time State as Feedback to Generate Test Cases. In: Proc. 29th International Conference on Software Engineering (ICSE'07), 2007.

[64] Daniel, B., Dig, D., Garcia, K. and Marinov, D. Automated Testing of Refactoring Engines. In: Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia, 2007. Pp. 185–194.

[65] Wang, Z., Elbaum, S. and Rosenblum, D. S. Automated generation of Context-Aware Tests. In: Proc. 29th International Conference on Software Engineering (ICSE'07), 2007.

[66] Pacheso, C. and Ernst, M. D. Eclat: Automatic Generation and Classification of Test Inputs. In: Proc. European Conference on Object-Oriented Programming (ECOOP'05), 2005. Pp. 504–527.

[67] Elbaum, S. and Diep, M. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. IEEE Transactions on Software Engineering, Vol. 31, No. 4, pp. 312–327, April 2005.

[68] Pacheco, C., Lahiri, S. K., Ernst, M. D. and Ball, T. Feedback-Directed Random Test Generation. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, USA, 2007.

[69] Pacheco, C., Lahiri, S. K. and Ball, T. Finding Errors in.NET with Feedback-Directed Random Testing. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA08), Seattle, Washington, USA, 2008. Pp. 87–95.

[70] Richardson, D. J., Aha, S. L. and O'Malley, T. O. Specification-Based Test Oracles for Reactive Systems. In: Proc. of the 14th International Conference on Software Engineering, Melbourne, Australia, 1992. Pp. 105–118.

[71] Kanstrén, T. Program Comprehension for User-Assisted Test Oracle Generation. In: Proc. 4th Int'l. Conf. on Software. Eng. Advances (ICSEA2009), Porto, Portugal, 2009.

[72] Antoy, S. and Hamlet, D. Automatically Checking an Implementation against Its Formal Specification. IEEE Transactions on Software Engineering, Vol. 26, No. 1, pp. 55–69, January 2000.

[73] Andrews, J. H. and Zhang, Y. General Test Result Checking with Log File Analysis. IEEE Transaction on Software Eng., Vol. 29, No. 7, pp. 634–648, July 2003.

[74] Ducasse, S., Gîrba, T. and Wuyts, R. Object-Oriented Legacy System Trace-Based Logic Testing. In: Proc. European Conference on Software Maintenance and Reengineering (CSMR'06), 2006.

[75] Roover, C. D., Michiels, I., Gybels, K., Gybels, K. and D'Hondt, T. An Approach to High-Level. In: Proc. 14th International Conference on Program Comprehension (ICPC'06), 2006, Behavioral Program Documentation Allowing Lightweight Verification.

[76] Xie, T. and Notkin, D. Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions. Journal of Automated Software Engineering, Vol. 13, No. 3, pp. 345–371, July 2006.

[77] Mesbah, A. and Deursen, A. van Invariant-Based Testing of Ajax User Interfaces. In: Proc. of the 31st International Conference on Software Engineering, Vancouver, Canada, 2009.

[78] Memon, A. and Xie, Q. Using Transient/Persisten Errors to Develop Automated Test Oracles for Event-Driven Software. In: Proceedings of the 19th International Conference on Automated Software Engineering, 2004.

[79] Haran, M., Karr, A., Last, M., Orso, A., Porter, A. A., Sanil, A. and Fouché, S. Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks. IEEE Transactions on Software Engineering, Vol. 33, No. 5, pp. 287–304, May 2007.

[80] Jin, H., Wang, Y., Chen, N.-W., Gou, Z.-J. and Wang, S. Artificial Neural Network for Automatic Test Oracles Generation. In: International Conference on Computer Science and Software Engineering, Wuhan, Hubei, 2008.Pp. 727–730.

[81] Bowring, J. F., Rehg, J. M. and Harrold, M. J. Active Learning for Automatic Classification of Software Behaviour. In: Proceedings of the International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, 2004. Pp. 195–205.

[82] Elbaum, S., Chin, H. N., Dwyer, M. B. and Jorde, M. Carving and Replaying Differential Unit Test Cases from System Test Cases. IEEE Transactions on Software Engineering, Vol. 35, No. 1, pp. 29–45, January/February 2009.

[83] Mackinnon, T., Freeman, S. and Craig, P. Endo-Testing: Unit Testing with Mock Objects. In: Proceedings of eXtreme Programming and Flexible Processes in Software Engineering – XP2000, Cagliari, Sardinia, Italy, 2000.

[84] Tillmann, N. and Schulte, W. Mock-Object Generation with Behaviour. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan, 2006. Pp. 365–368.

[85] Saff, D., Artzi, S., Perkins, J. H. and Ernst, M. I. D. Automatic Test Factoring for Java. In: Proceedings of the 20th Annual IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, 2005, pp. 114–123.

[86] Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R. and Majumdar, R. Generating Tests from Counterexamples. In: Proceedings of the 26th International Conference on Software Engineering, 2004. Pp. 326–335.

[87] Pesonen, J. Extending Software Integration Testing Using Aspects in Symbian OS. In: Proceedings of Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC-PART 2006), 2006. Pp. 147–151.

[88] Bertolino, A., Angelis, G. De, Lonetti, F. and Sabetta, A. Let the Puppets Move! Automated Testbed Generation for Service-Oriented Mobile Applications. In: Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications, Parma, Italy, 2008.Pp. 321–328.

[89] Binder, R. V. Testing Object-Oriented Systems – Models, Patterns and Tools. Addison-Wesley, 1999.

[90] Puolitaival, O.-P., Luo, M. and Kanstrén, T. On the Properties and Selection of Model-Based Testing Tool and Technique. In: Proceedings of the 1st Workshop on Model-Based Testing in Practice (MoTiP2008), Berlin, Germany, 2008.

[91] Neto, A. D., Subramanyan, R., Vieira, M., Travassos, G. H. and Shull, F. Improving Evidence about Software Technologies: A Look at Model-Based Testing. IEEE Software, Vol. 25, No. 3, pp. 10–13, May/June 2008.

[92] Ramamoorthy, C. V., Ho, S. F. and Chen, W. T. On the Generation of Program Test Data. IEEE Transaction on Software Engineering, Vol. 2, No. 4, pp. 293–300, 1976.

[93] Blackburn, M., Busser, R., Nauman, A., Knickerbocker, R. and Kasuda, R. Mars Polar Lander Fault Identification Using Model-Based Testing. In: Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS02), 2002. Pp. 163–169.

[94] Bringmann, E. and Krämer, A. Model-Based Testing of Automotive Systems. In: Proceedings of the International Conference on Software Testing, Verification and Validation, 2008. Pp. 485–493.

[95] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R. and Stauner, T. One Evaluation of Model-Based Testing and its Automation. In: Proceedings of the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, 2005. Pp. 392–401.

[96] Vieira, M., Song, X., Matos, G., Storck, S., Tanikella, R. and Hasling, B. Applying Model-Based Testing to Healthcare Products: Preliminary Experiences. In: Proceedings of the 30th International Conference on Software Engineering (ICSE08), Leipzig, Germany, 2008. Pp. 669–671.

[97] Chikofsky, E. J. and Cross II, E. J. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, Vol. 7, No. 1, pp. 13–17, 1990.

[98] Storey, M.-A. Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future. Software Quality Journal, Vol. 14, No. 3, pp. 183–208, September 2006.

[99] Cornelissen, B., Zaidman, A., Deursen, A. van, Moonen, L. and Koschke, R. A Systematic Survey of Program Comprehension through Dynamic Analysis. IEEE Transaction on Software Eng., 2009.

[100] Pollari, M. and Kanstrén, T. A Probe Framework for Monitoring Embedded Real-Time Systems. In: Proc. 4th International Conference on Internet Monitoring and Protection (ICIMP 2009), Venice/Mestre, Italy, 2009. Pp. 109–115.

[101] Kanstrén, T., Piel, E. and Gross, H.-G. Observation Based Modeling for Model-Based Testing. Submitted to Journal of Software Testing, Verification and Reliability, 2009.

[102] Biggerstaff, T., Mitbander, B. G. and Webster, D. E. Program Understanding and the Concept Assignment Problem. Communications of the ACM, Vol. 37, No. 5, pp. 72–82, May 1994.

[103] Antoniol, G. and Guéhéneuc, Y.-G. Feature Identification: An Epidemiological Metaphor. IEEE Transactions on Software Engineering, Vol. 32, No. 9, pp. 627–641, September 2006.

[104] Rohatgi, A., Hamou-Lhadj, A. and Rilling, J. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. In: Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008. Pp. 236–241.

[105] Robillard, M. P. and Murphy, G. C. Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology, Vol. 16, No. 1, February 2007.

[106] Mariani, L., Papagiannakis, S. and Pezzé, M. Compatibility and Regression Testing of COTS-Component-Based Software. In: Proc. 29th International Conference on Software Engineering (ICSE'07), 2007.

[107] Lorenzoli, D., Mariani, L. and Pezzè, M. Automatic Generation of Software Behavioral Models. In: Proceedings of the 30th International Conference on Software Engineering (ICSE08), Leipzig, Germany, 2008. Pp. 501–510.

[108] Xie, T. and Notkin, D. Automatic Extraction of Object-Oriented Observer Abstractions from Unit-Test Executions. In: Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM04), Seattle, Washington, USA, 2004. Pp. 290–305.

[109] Cook, J. E. and Du, Z. Discovering Thread Interactions in a Concurrent System. Journal of Systems and Software, Vol. 77, No. 3, pp. 285–297, September 2005.

[110] Cook, J. E. and Wolf, A. Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 3, pp. 215–249, 1998.

[111] Walkinshaw, N., Bogdanov, K., Ali, S. and Holcombe, M. Automated Discovery of State Transitions and their Functions in Source Code. Software Testing, Verification and Reliability, Vol. 18, No. 2, pp. 99–121, June 2008.

[112] Parsons, T., Mos, A., Trofin M., Gshwind, T. and Murphy, J. Extracting Interactions in Component-Based Systems. IEEE Transactions on Software Engineering, Vol. 34, No. 6, pp. 783–799, November/December 2008.

[113] Briand, L. C., Labiche, Y. and Leduc, J. Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. IEEE Transactions on Software Engineering, Vol. 32, No. 9, pp. 642–663, September 2006.

[114] Bennett, C., Myers, D., Storey, M.-A., German, D. M., Ouellet, D., Salois, M. and Charland, P. A Survey and Evaluation of Tool Features for Understanding Reverse-Engineered Sequence Diagrams. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 20, No. 4, pp. 291–315, July 2008.

[115] Marburger, A. and Westfechtel, B. Tools for Understanding the Behavior of Telecommunication Systems. In: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, 2003. Pp. 430–441.

[116] Schmerl, B., Aldrich, J., Garlan, D., Kazman, R. and Yan, H. Discovering Architectures from Running Systems. IEEE Transactions on Software Engineering, Vol. 32, No. 7, pp. 454–466, July 2006.

[117] Aalst, W. M. P. van der, Rubin, V., Verbeek, H. M. W., Dongen, B. F. van, Kindler, E. and Günther, C. W. Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. Software and Systems Modeling (SoSyM), 2009.

[118] Aalst, W. M. P. van der, Dongen, B. F. van, Günther, C. W., Mans, R. S., Alves de Medeiros, A. K., Rozinat, A., Rubin, V., Song, M., Verbeek, H. M. W. and Weijters, A. J. M. M. ProM 4.0: Comprehensive Support for Real Process Analysis. In: Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN07), Siedlce, Poland, 2007.

[119] Lo, D., Khoo, S.-C. and Liu, C. Mining Temporal Rules for Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 20, No. 4, pp. 227–247, 2008.

[120] Lo, D., Mariani, L. and Pezze, M. Automatic Steering of Behavioral Model Inference, Amsterdam, Netherlands, 2009, Proc. 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009).

[121] Koskinen, J., Kettunen, M. and Systä, T. Profile-Based Approach to Support Comprehension of Software Behavior. In: Proceedings of the 14th International IEEE Conference on Program Comprehension (ICPC06). Pp. 212–224.

[122] Jacky, J., Veanes, M., Campbell, C. and Schulte, W. Model-Based Software Testing and Analysis with C#. Cambridge University Press, 2008.

PAPER I

Integrating and Testing a System-Wide Feature in a Legacy System

An Experience Report

In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR'07, Amsterdam, the Netherlands, 21–23 March, 2007. 10 p.
© 2007 IEEE.
Reprinted with permission from the publisher.

Integrating and Testing a System-Wide Feature in a Legacy System: An Experience Report

Teemu Kanstrén, Mika Hongisto, and Kari Kolehmainen
VTT Technical Research Centre of Finland
P.O.Box 1100, FI-90571 Oulu, Finland
{teemu.kanstren, mika.hongisto, kari.kolehmainen}@vtt.fi

Abstract

This paper describes our experiences with integrating and testing an embedded, system-wide feature called Dynamic Voltage and Frequency Scaling (DVFS) in a software platform for mobile devices. DVFS affects the whole system by scaling the hardware performance levels during run-time. Implementing and testing the basic functionality of DVFS was easy, however verifying that the whole system worked after integration was more difficult. The platform was legacy code which had not been developed with any consideration for this kind of a feature. We had to consider the complex run-time behaviour of the whole platform, including operating system services, device drivers and applications. DVFS could cause problems and failures in almost any part of the system. Based on our experiences, we describe problems in integrating and testing a system-wide feature like DVFS and suggest possible directions for future research to help address some of these problems.

1. Introduction

The role of software testing in general can be defined as exercising the system under test (SUT) with different inputs in order to reveal possible errors [3][6][23]. This usually refers to testing an application, component or some functionality of a system on its own or as a part of a larger context. These components and functionality are usually considered as something testable on their own, decoupled from the system. When making a change into a software system a common practice is to try to localize the change as much as possible to enable testing the software in smaller parts. This requires preventing the ripple effect where the change cascades to other parts of the system. However, with system-wide features that affect the whole platform this becomes more difficult.

When a new integrated feature is system-wide, changes can not be localized and the ripple effect can not be prevented. In fact, this produces the ultimate ripple effect by affecting the whole system and all of its parts. Testing the feature as a part of the system in this situation is more complex as we need to consider all the behaviour, parts and interactions in the system. Additional complexity is added by the fact that these days software is developed in an increasingly collaborative fashion, integrated from components that are provided by different companies and sometimes only available as binary executables. These types of features and systems provide new and different problems to consider in integration and testing.

In this experience report we describe our experiences with Dynamic Voltage and Frequency Scaling (DVFS), a system-wide embedded feature in a software platform for mobile devices. DVFS affects the whole system by scaling the performance levels during run-time, with the goal of producing savings in power consumption. Implementing and testing the basic functionality of DVFS was straightforward, however verifying that the whole system worked after integrating it was far more challenging. The platform was legacy code which had not been developed with any consideration for scaling the system voltage and CPU speed. We had to consider the complex run-time behaviour of the whole system, including operating system services, device drivers and applications. The cause of problems and failures could be anywhere in the system. This includes components provided by different companies, some only as binary executables. Based on our experiences with DVFS we describe the problems involved in integrating and testing a system-wide feature in a legacy system and suggest directions for possible future research.

Section 2 describes the DVFS system from both the hardware and software viewpoints. Section 3 describes our experiences in implementing, testing and debugging DVFS. In section 4 we consider the causes

11th European Conference on Software Maintenance and Reengineering (CSMR'07)
0-7695-2802-3/07 $20.00 © 2007 IEEE

of these problems and in section 5 we highlight some key research areas to help solve make this easier. Section 6 concludes the paper.

2. Dynamic Voltage and Frequency Scaling

DVFS requires careful consideration of both hardware and software components in co-operation. In this section, we first describe the hardware related parts and their dependencies to give some background and motivation for how the software needed to be designed. Second, the DVFS related software components and their roles in the system are described. A detailed description of DVFS concepts can be found from [15].

2.1. Hardware

The core of the hardware platform we were working with was Texas Instruments OMAP2420 multimedia processor. OMAP2420 is system-on-chip product that has several integrated processors and features, including support for Dynamic Voltage Scaling (DVS) that was required for our research and development work.

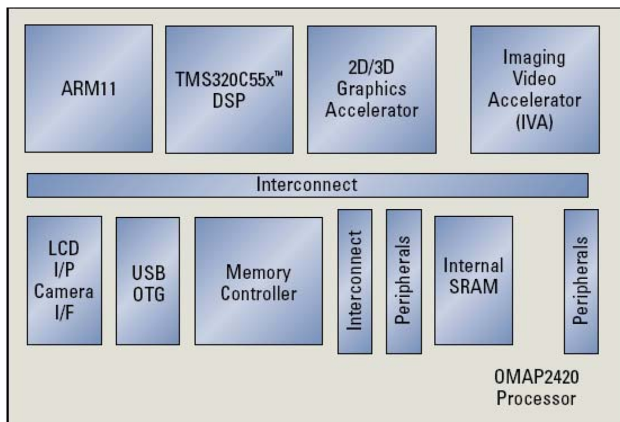


Figure 1. OMAP2420 architecture [26].

High abstraction of OMAP2420 processor and interconnect architecture is shown in Figure 1. Starting from processors: Texas Instruments TMS320C55x (DSP) provides audio processing, ARM11 Family ARM1136JS (MPU) provides general purpose processing, Imagination PowerVR MBX (GFX) provides 2D/3D graphics acceleration [17] and dedicated Imaging and Video Accelerator (IVA) provides video encoding and decoding [26]. A high speed shared interconnect bus provides communication between processors and memory. A peripheral interconnect bus provides communication for less data intensive peripherals.

With OMAP2420 it is possible to halt the processing of processors and change operation voltage of the whole chip [27]. The voltage changing process is shown in Figure 2. Whenever voltage is changed, it is done through voltage meta state where Dynamic Memory Access (DMA) transfers are completed and halted, processing units are halted, interrupts are disabled and hardware parameters are reconfigured.

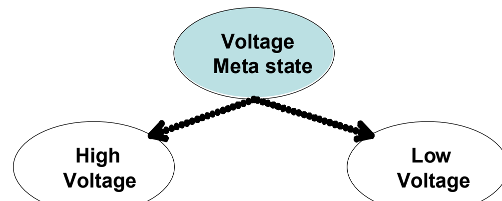


Figure 2. Voltage adjustment process.

As OMAP2420 provides only one voltage domain [25], voltage change affects the whole domain and all hardware components in it. As some components require higher voltage than others, the voltage needs to be carefully managed. Clock speed of most of the processors can be scaled independently of each other. Lowest possible operation voltage should be used for selected performance level for each processor to maximise energy efficiency. This is not always possible, as processing load is not evenly distributed over processors [15].

Additional difficulty is hardware characteristics. Some peripherals and processors might require high voltage to function properly regardless of the configured clock speed. Activity of some peripherals can also prevent voltage scaling process, and thus, needs to be considered before they are enabled. These characteristics require system wide awareness of what is going on in hardware and what is needed by software.

All voltage and clock requirements of processors and peripherals need to be taken into account when developing device drivers and hardware resource management. For example, a typical behaviour of a device driver: direct access into hardware registry to enable clock domains of processor. This can cause a system crash if the provided voltage is too low. Centralised and protected control of hardware parameters is necessary to provide a stable system.

Even though the hardware allows several different performance levels and independent scaling of processor clock speeds, we settled for two operation points: Full clock speed with high voltage and half clock speed for decreased voltage. This provides a reasonable trade off between development effort (especially configuration and testing work) and gained savings in energy consumption.

2.2. Software

The software platform we were working with is based on top of the Symbian operating system. The code base has evolved over a course of more than a decade and thousands of engineers have taken part in developing it. The software platform consists of hundreds of components, provided by many different companies. The total size of the code for the system we worked on is close to 20M physical SLOC, mostly C and C++. This is the number of source lines of code after removing blank white space and comments [24]. In addition to this the system contains a number of third-party commercial off the shelf (COTS) and similar components that are only available in binary format and not included in the SLOC count. These binary only components include many of the components we had to work with. From the DVFS viewpoint the system was all legacy code as it had not been designed with any consideration for this type of feature. The functionality of DVFS needed to be integrated with the large existing code base, including the third-party components.

DVFS is implemented in the system as a resource which the different components can reserve when they need a certain performance level. These performance levels are called Operation Points (OP), and in our case we had two OP's, a high and a low OP. The DVFS resource along with several other resources is handled by a system Resource Manager component. When there are one or more reservations is the system for the DVFS resource, the hardware is set to high OP. The correct functionality of the DVFS required two different types of components to be directly DVFS aware: system load monitoring components and device drivers. System load monitoring is needed to reserve the high OP when the system performance requirements rise higher than what is provided by the low OP. Reservations must also be made when a hardware component is used by an application or OS service that requires the higher OP to function correctly. These hardware components are used through their device drivers or similar components and thus these components must make the reservation when required. When these components no longer need the higher OP, they must release their reservation.

As the DVFS functionality is at the very core of the system, it was implemented as a Symbian OS kernel extension. The core implementation of the DVFS feature consists of the integration of many related and affected components in the system:

1. The Resource Manager.
2. System load monitor and performance tuner.
3. Interface to system load information.
4. Glue components to integrate the load monitor with the Resource Manager and the system load information interface.
5. Device drivers for the hardware components that need to reserve high OP when used.

These central DVFS components and their relationships are shown in Figure 3, which shows how the DVFS implementation is not a single component but a product of integration of many components. Only the glue code to integrate the components and some of the device drivers are in-house products. All other components are from third-parties and each of these from a different company. The third-party components were integrated with glue code, while the in-house drivers were directly changed to work with the third-party components. In addition, many other components in the system are affected through their use of the drivers or changes in the system performance level.

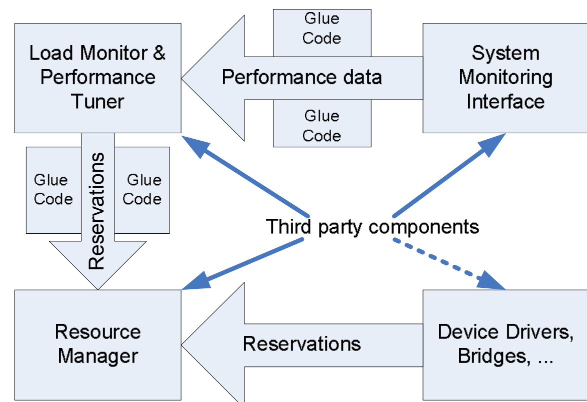


Figure 3. The central DVFS components.

In addition to having direct effect on these core components, DVFS also affects other parts of the system. Since the kernel of the operating system is a real-time operating system, it must meet real-time timing constraints. For example, when playing MP3 music on the device, the current OP of the system and changing it must have no noticeable effect for the user. Thus the load monitoring of the system must keep the performance stable and high enough not to cause problems for the users and the user experience should be equal to running in high OP all the time.

Another property that also affects all parts of the system is changing the hardware parameters. The change to low operation point requires reconfiguring some of the hardware components at runtime and if all values are not set correctly, any functionality that makes direct or indirect use of this hardware will fail.

For example, the SDRAM memory parameters with different hardware configurations are different in low and high OP and if they are not set correctly certain memory accesses can fail the system. DVFS should be transparent to as much of the system as possible, as we can not expect all components to take DVFS into account. Thus it is the responsibility of the DVFS implementation and the dependent components to make these configurations when the OP is changed. For us this meant we needed to find how and where DVFS affects the system and verify these cases are handled correctly.

Figure 4 shows a high level overview of how the DVFS affects the system. As described earlier, the device drivers must be DVFS aware and reserve the required OP when used. The applications in the system are affected by the performance level in use and their performance requirements handled by the load monitor. Finally, the hardware parameters must be configured correctly for each OP since misconfiguring these produces instability in all parts of the system, including the device drivers and applications.

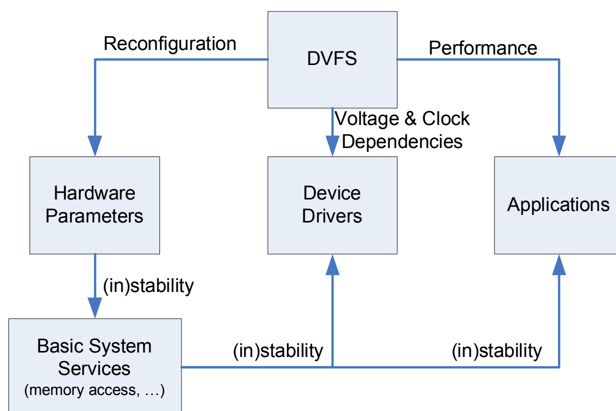


Figure 4. DVFS effects on the system.

Variability in the platform causes some products to implement the DVFS functionality differently. Some use DVFS while others may only use Dynamic Frequency Scaling (DFS). This requires variation in the device drivers. Some products can also have several operation points while others will only have a high and a low operation point. From the software point of view this means that the Resource Manager needs to implement different functionality when the products using the software platform make use of different types of resources. Also the glue code component to integrate the load monitoring component to the system needs to change when there are different numbers of operation points in the system and the set of hardware parameters to configure varies as some are dependent on the scaling of voltage and others on the frequency. When underlying hardware is changed, the drivers and their interactions can also be different.

3. Problems Encountered

We identify the following main problems in integrating, testing and debugging DVFS: architectural mismatches in integration, identifying the DVFS related dependencies in the system, verifying the dependent components, observing the state of the system, finding the cause of failures and debugging binary components.

Architectural mismatches in integration were found even though the integrated components were built with the same system in mind. All the core DVFS components were designed and tested to work in the same system by their providers. However, mismatches still arose as they were only tested together in integration. For example, the Resource Manager was not thread-safe, meaning when it was used from several concurrent threads at the same time its internal state would become corrupt. Yet the integrated components used it in a concurrent manner. Thus the calls to it needed to be synchronized externally. However, the load monitor used Symbian nanokernel threads to run itself, and the Symbian synchronization objects such as semaphores and mutexes would not work on nanokernel threads but required higher level user threads to work. These problems point to similar architectural mismatches as described by Garlan et al. [13] who studied the integration of several separate COTS style components. In this case it is interesting that these components were developed with the same system in mind, yet they still set mismatching assumptions about their environment. Getting these components to work together as such was not possible and required getting the third parties to make changes to their components, which was often a slow and time consuming effort. This also requires effectively pinpointing the cause of failure in several third party components before getting it fixed.

Identifying the DVFS related components was easier for some components and more difficult for others. This was the first time this type of feature was being implemented in the platform and it was not clear what were the exact performance levels required by each part of the system. When a component was known to need a higher performance level because of hardware limitations, it was clear that it needed to reserve the high OP when it was used (to provide necessary clock or voltage). However, for all components this was not so clear. When the system would fail we had to consider the possibility of the required resources not having been reserved. And we could only discover this

11th European Conference on Software Maintenance and Reengineering (CSMR'07)
0-7695-2802-3/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

by monitoring the system execution at the time of failure, considering the involved components and if any were involved that could require higher OP. As there were many potential causes of failure and we did not know in advance all components that needed the high OP and when, it was not possible to know immediately if a failure was caused by a mishandled dependency or something else.

A second issue with DVFS related dependencies were the applications running on top of the OS. Since DVFS affects the system performance levels (CPU MHz), it also affects the processing power available to these applications. Since the system was only implemented to monitor CPU load, applications that require only different type of performance from the hardware will not perform optimally. For example, reading from file system can generate high load on the hardware buses but not enough load on the CPU to make the load monitor raise the performance level. As bus operations are dependent on the system clock speed, the performance is not optimal. Also applications causing fluctuating load can cause problems for the system when the OP changes continuously. Knowing all these cases in advance is not possible as there are too many possibilities.

Verifying the DVFS dependent components proved to be a problem as some did not seem to correctly request and release the high OP when needed. The easy route was to blame the driver developers. However, the problem could also be in other components using the drivers and not freeing all resources, which caused the driver not to release its reservations. As without DVFS in the system many of these cases would not show up as problems, they were not noticed before. Thus adding DVFS into the system would make the driver seem faulty even though the problem could be elsewhere. Also, without means to effectively identify every DVFS dependency in the system, even knowing all required drivers in advance was not possible.

Finding the cause of failures was problematic especially when the cause was in wrongly configured hardware parameters. For example, when the SDRAM memory parameters for the lower OP were wrong the system would hang in seemingly random functionality after changing to low OP. For example, we received a bug report for the USB driver not working with the DVFS, where the real cause was actually a misconfigured memory parameter. At the same time we were also experiencing problems in many other parts of the system, which later turned out to be for the same reasons. To make things more complicated, the memory parameters were configured by a third party component, who insisted the parameters should be similar on our platform as on theirs. Thus we did not consider this as a possibility of failure at first. Only after long debugging sessions did we come to think of them as a possible cause of failure and fix it, as in truth we found there was a small difference between our platform and that on which the component provider tested and configured the parameters.

Observing the state of the system was difficult. At a time of failure the system would usually just crash with no clear indication of what went wrong. Typically this would not produce any trace of execution as is typical for embedded systems. To get some idea of what was happening inside the system, we instrumented the kernel to show which thread was executing at which time. However, even when we had some idea of where the execution was when it ended, it was not always very useful. The true cause was often somewhere else in the system such as wrong hardware parameters, a change in system state earlier in the system execution or a delayed function call (DFC) started from some other part of the system. Finding errors in system behaviour and debugging the faults was then complicated by the lack of visibility into the system and the availability of detailed trace data.

Debugging binary components is always more difficult and our case was no different. In addition, with DVFS we could not just forcus on a single binary component but also had to consider the fact that the cause of failure could be anywhere in the system. For example, when the system was changing from one OP to another, many of the operating system services (such as DMA) had to be stopped for the time of the change and restarted after. Since many of the services were dependant on other services, these had to be stopped and restarted in the correct order. This was handled by each dependent component having a pre- and post-change notification function that was called before and after the OP was changed and in which they handled the stop and restart functionality. These components each had an order number they had to set which determined the order in which they would be notified. However, since many of these components were provided by different companies, some of them set mismatching notification orders. When the system would hang on changing the OP, knowing if the problem was in these components or somewhere completely different was difficult since we could not directly see the order numbers in the binary components. Instead, we had to infer them through instrumenting the Resource Manager and through other similar methods.

A second concept related to debugging of binary components is when these components are composed of many different components themselves. For example, we had a case where one of the driver components did not seem to be working correctly by not releasing resources when needed. This actually

11th European Conference on Software Maintenance and Reengineering (CSMR'07)
0-7695-2802-3/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

turned out to be a problem in another component provided with the driver, not in the driver code itself. The component would reserve a communication channel to the driver, which resulted in the resource being kept reserved. It should have released the channel when done but did not. Since the resource was not previously needed, the fault in the component using the driver was not visible earlier. This shows how a third-party component itself can be made up of several components, which again need to be debugged. If the failure is only visible in integration, the integrator has to locate and fix failures in all these components.

In summary, Figure 5 shows a simplified view of how the DVFS related dependencies and possible failure causes spread in the system. For clarity only a few of each type of arrows are shown, whereas in reality many more dependencies exist between the components. Some of the dependencies are temportal, for example restarting the basic system services need to be handled in correct order when changing the OP. Many components are only available in binary form and made by different companies.

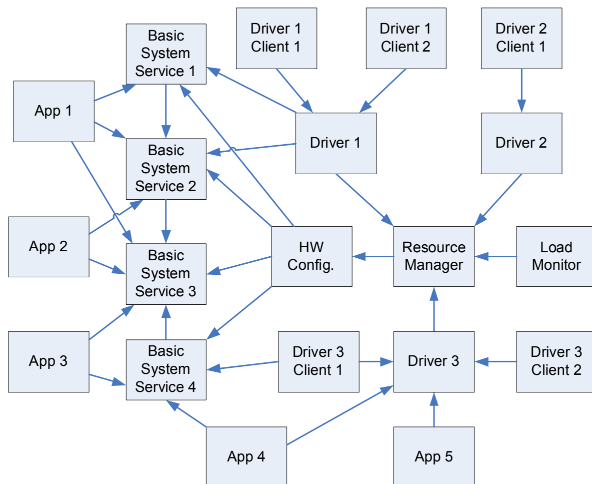


Figure 5. Overall DVFS dependency view.

4. Problem Causes

Considering the problems we had, some could be considered failures to adhere to good software development practices. However, when new system-wide features are introduced into a legacy system they also give rise to specific problems. Dependencies and effects spread across the components, with little possibilities to affect this. Through these components, the dependencies also affect and require the attention of all the companies involved in the collaborative development effort. Based on our experiences, we classify the problem causes to three categories: *collaboration processes*, *testability* and *debugging*. Each of these is considered in this section.

4.1. Collaboration Processes

Modern software systems are being developed in an increasingly collaborative fashion, from in-house components, commercial off the shelf (COTS) components and specifically tailored third party components. Large parts of development are outsourced and software is developed in different collaboration models together with other companies [20]. As system-wide features affect large parts and many components at once, this affects and requires co-operation of many component providers. As an example of this, DVFS is a single feature in a software platform for mobile devices yet it was made up of and integrated with components from several companies.

To us, the mismatches and problems in addressing them when integrating components developed for the same system highlights many of these collaboration aspects. When a system is developed in a collaborative fashion from components provided by many companies, the collaborators often do not have the whole system or all components to test their component with. Thus the integration becomes solely the integrators responsibility, who often does not have detailed knowledge of individual components. When a new system-wide feature is introduced into a system, it will also create ripples over the system and require the collaborative work of all the involved component providers to fix the issues. When the work is specified to detail in contracts, addressing these issues effectively becomes long, slow and difficult. Instead, fast responsiveness and a more evolutionary approach are needed. The companies we worked with included samples of companies closer to each extreme. Some of the issues we faced could only be effectively resolved when a collaborator company was willing to make the required changes quickly. In practice this meant in the period of several weeks, not several months.

4.2. Testability

In the past hardware testability has received more attention than software testability [5][12]. However, lately also software testability has been receiving a growing interest [5][8][11][19][22][29]. Although DVFS is a closely hardware related feature, our viewpoints are on the software testability side. Software testability in the literature has varying definitions. Our view of testability in this section is in line with Binder [5] and is concerned with the effort

11th European Conference on Software Maintenance and Reengineering (CSMR'07)
0-7695-2802-3/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

and ease of testing the components and their functionality. Another related concept is that of Jungmayr [19] who relates testability to identifying test-critical dependencies. These are the dependencies most critical for achieving a good testability of a system.

Observability and controllability are considered by many as basic properties of testability [1][5][12][19]. To test a component we must be able to control its input (and internal state) and observe its output (and internal state) [5]. For effective testing, we need to be able to control the spread of code execution and observe the results. This is easiest when dealing with white-box components, where we have the source code available and can modify it for testability support. Figure 6 shows some possible aspects of testability in such a case.

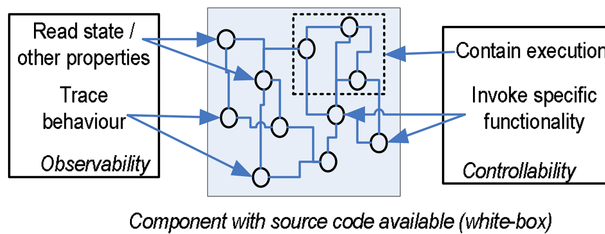


Figure 6. White-box testability.

In this case, if we can modify the code, we can add support for testability. This allows breaking dependencies by designing the code as independent units, making it possible to insert stubs, mocks or similar objects and using other similar techniques. Visibility into component internals allows for example invoking specific functionality, reading states at different times and adding tracing code. This can be limited by the company policies and system size, which affect how much it is possible to actually influence the design of the components for which we have the source code available. For example, when implementing our own components related to DVFS, the techniques presented by Feathers [11] for breaking dependencies to bring legacy code under test were of great help. However, since we did not have control over most of the platform, we were only able to apply these techniques to small parts of the code.

With third party binary components we are dependent on what is provided by the component provider as shown in Figure 7. The component is a black box into which we only see through the interfaces provided. We can not modify the components code, contain execution or see internal states, unless supported by the component through specialized interfaces. These interfaces are likely to be specific to some given task which the component provider has specified for the component. As it is not possible to know all problems and testing requirements that will be met in the course of integration, the likeness of a component providing required interfaces is not high. In our case we did not have interfaces for all the information we would have needed, and could not ask for them to be included in advance as we did not know the effects on all the components in advance. When the components are already delivered, getting new interfaces is difficult. Some observability can be achieved by monitoring at the platform level such as when we instrumented the kernel to provide traces of thread execution. However, the information available at this level is very coarse grained and not always very helpful in debugging a specific problem.

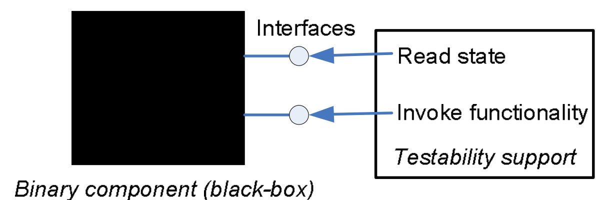


Figure 7. Black-box testability.

In real systems the different types of components are combined together as shown in Figure 8. This is where the effects of a system-wide feature such as DVFS are most visible. We had to consider the interactions of all the parts in the system together to verify DVFS. Our options to affect the system were limited as when we had the source code for the components, we could see their internals and add tracing code, but could not permanently modify them for testability. As the components involved also included a number of black box components, getting an overview of what was happening inside the system and where the problems were was difficult.

When we add new system-wide functionality to a large system, we need to understand the diverse interactions and how they are affected. This is difficult on its own but more difficult when it is not possible to get a good view of the involved components, how they act and what is their state at a given time. We could see how a single component works and when it passes control to another component. However, what happened after passing control to a black box was not visible. At the same time it is difficult to get a good understanding of the complex interactions and relations inside the whole system, with concurrent threads executing code from many involved components. We had to do much of this analysis and tracing manually, which becomes difficult as the amount of data to analyse and trace quickly grows very large.

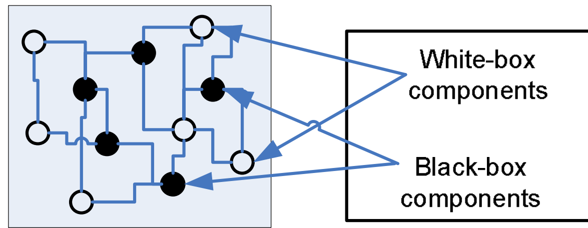


Figure 8. Testability on a system level.

4.3. Debugging

Debugging the system is closely coupled with the software testability. To debug something we first need to have a test to find the fault that we will be debugging. Without automated tests to verify a fix, the process of fixing failures becomes tedious and slow. Debugging has received a lot of research interest where much recent work has focused on finding the cause of failures from program execution. These techniques focus on such issues as isolating failure-inducing input [21][30], code changes [9] and debugging of failed test cases [2] [14][18].

However, our problems were different in nature. We did not have any specific set of input we could vary for the system to see which would break it. The problems we had were not introduced by any specific code changes but rather a system wide change of performance level, the ripples of changes it caused and the integration of ready-made components, including some in only binary form. The feature affected many components in the system, all which had to work together in complex ways and considered as potential causes of failure. In many cases such as when changing hardware parameters, the failures would only show up as side effects in many different components. As described earlier, we could not easily decouple testable parts from the system and thus had no automated tests to verify the faults. We had to rely widely on slow and ineffective manual testing.

When debugging a system we are trying to find the cause of failures when we already know something is not working. This requires the ability to observe the internal states and behaviour of the relevant parts of the system. To enable this, first the relevant parts of the system and the information of interest must be identified, which in a system-wide feature can require analysing large amounts of data and the interactions of large parts of the system. When the feature and thus its failure is not localized, we need to be able to monitor the system state as a whole and find the possible changes in state or behaviour that could be causing the failures. In most of the cases we had, there were many possible causes of failure and they could all cause problems in many different parts of the system. As we did not have effective techniques to monitor and analyse the system at this level, finding and fixing a cause of failure often came down to long debug sessions including making educated guesses, digging information from the system and trying different possibilities.

5. Future Directions

In this paper we described our experiences in integrating, testing and debugging DVFS, a system-wide feature of an embedded system with a large legacy code base. This proved problematic as the system was never designed with such a feature in mind and the feature affected the whole system and all of its parts. Yet more difficulties were brought by the fact that the system contained components from various companies, some only in binary form, which all had to be considered.

We expect system-wide features to become more common in complex software intensive systems. In embedded systems the close coupling of software with hardware makes them possible as in the case of DVFS. In software systems in general, the modern software development techniques and platforms such as aspect oriented programming and the possibilities provided by virtual machine environments will provide ample possibilities for similar features and problems. Many of the issues we faced are related to designing the systems and components with testability in mind. To this end we suggest the following research topics as helpful as related to integration and testing of these features in modern environments:

1. *Light weight, agile collaboration models.* As systems are made up of increasing number of components from different parties, changes affect more collaborators and require effective, responsive collaboration models. Research into collaborative software development is ongoing and while different collaboration models are considered [20], they still need more work and a wider adoption to effectively address the cases where changes affect many collaborators at once.
2. *Identifying feature dependencies and testing parameters in a system.* Identifying actual dependencies in a system is needed for effective re-engineering of a system. This is important for both integration work and for identifying the relevant parameters for testing the feature. DVFS affects some very basic properties of the system

and finding the affected components requires analysing large amounts of dynamic software runtime-behaviour. This needs to be simplified. We believe possible techniques to assist in this include visualisation [28] and modelling [16] of system state and behaviour, and behaviour classification [7][10].

3. *Improving the testability of legacy code in constrained environments.* Feathers [11] provides a catalogue of techniques to assist in bringing legacy code under test. However, these existing methods are mostly based on the assumption of being able to access and modify the source code of involved components. When the development happens in a collaborative context and components are provided only in binary form, control over these components is limited. Thus techniques for this type of an environment where we have less control over changes are needed.
4. *Testability at the system level including binary components.* In system-wide features the cause of the problem as well as its symptoms are often spread over the system in various components and focusing on a single component will not reveal the errors. Effectively addressing this issue requires gathering and analysing data from the system execution and states as a whole. Though different in nature, as possible research topics we suggest similar topics as for identifying feature dependencies in system state: tracing, classifying, modeling and visualising software states and behaviour. For binary components this also requires special support from the components or monitoring support from the platform itself. Generic approaches that make it possible to address internals of binary components are needed. These topics are ongoing in component based systems research [4] but still need more support for especially in system-wide context.

6. Conclusions

When new features to a software system are planned, they are often considered as just another independent feature in the system. We started with thinking DVFS as just scaling the voltage and frequency to save some power. However, with its system-wide effects and dependencies, we quickly learned differently. Some features are more independent than others, while some are more coupled. DVFS is on the far end of coupling with its system-wide effects. In this paper we described our experiences in integrating and testing DVFS into the system. We identified the following problem areas:

- Changes affecting components and requiring attention of many collaborators at a same time
- Identifying system-wide dependencies
- Accessing the states and properties at system level and in binary only components
- Controlling the system execution
- Finding the causes of failures from the large data sets and traces

We believe integrating and testing this type of features is and will remain a challenging topic. However, we also believe this work can be made easier by addressing these issues with better techniques, tools and methods.

10. References

[1] B. Baudry and Y.L. Traon, "Measuring design testability of a UML class diagram", *Journal of Information and Software Technology 47*, 2005, pp. 859-879.

[2] B. Baudry, F. Fleurey and Y.L. Traon, "Improving Test Suites for Efficient Fault Localization", *Proc. 28th Int'l. Conf. Software Eng. (ICSE'06)*, 2006.

[3] B. Beizer, *Black Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, Inc., 1995, 320 pp.

[4] S. Beydeda and V. Gruhn, "State of the art in testing components", *Proc. 3rd Int'l. Conf. Quality Software 2003*, Nov., 2003, pp. 146-153.

[5] R. Binder, "Design for testability in object-oriented systems", *Communications of the ACM*, Vol. 37, No. 9, Sept. 1994, pp. 87-101.

[6] R. Binder, *Testing Object Oriented Systems*, Addison-Wesley, Reading, MA, 2000. 1200pp.

[7] J.F. Bowring , J.M. Rehg and M.J. Harrold, "Active learning for automatic classification of software behaviour", *Proc. Int'l. Symposium on Software Testing and Analysis (ISSTA 2004)*, July 2004, pp 195–205.

[8] M. Bruntik, A. and Van Deursen, "Predicting Class Testability Using Object-Oriented Metrics", *Proc. 4th IEEE Int'l. Workshop on Source Code Analysis and Manipulation (SCAM'04)*, 2004, pp. 136-145.

[9] O.C. Chesley, X. Ren and B.G. Ryder, "Crisp: A Debugging Tool for Java Programs", *Proc. 21st IEEE Int'l. Conf. on Software Maintenance (ICSM'05)*, 2005.

[10] W. Dickinson, D. Leon and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles", *Proc. 23rd Int'l. Conf. Software Eng.(ICSE 2001)*, Toronto, Ontario, Canada, 2001, pp. 339-348.

[11] M. Feathers, *Working Effectively With Legacy Code*, Prentice Hall, 2005, 304 pp.

[12] S. Freedman, "Testability of Software Components", *IEEE Trans. Software Eng.*, vol.17, no.6, June 1991, pp.553-564.

[13] D. Garlan, R. Allen and J. Ockerbloom, "Architectural Mismatch or Why it's hard to build systems out of

0-7695-2802-3/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

existing parts", *Proc. 17th Intl. Conf. Software Eng. (ICSE'95)*, 1995, pp. 179-185.

[14] M. Gälli, M. Lanza, O. Nierstrasz and R. Wuyts "Ordering Broken Unit Tests for Focused Debugging", *Proc. 20th IEEE Int'l. Conf. on Software Maintenance (ICSM'04)*, 2004.

[15] M. Hongisto and K. Kolehmainen, "Dynamic Voltage Scaling Framework for Mobile Multimedia Systems", *Proc. 4th IASTED Int'l. Conf. Communications, Internet and Information Technology (CIIT 2006)*, 2006.

[16] J. Huselius, and J. Andersson, "Model Synthesis for Real-Time Systems", *Proc. 9th European Conf. Software Maintenance and Reengineering (CSMR 2005)*, 2005.

[17] Imagination Technologies Ltd., "Texas Instruments Announces OMAP 2 Devices Utilising PowerVR MBX Graphics Core", *press release*, Feb. 2004, http://www.imgtec.com/news/Release/index.asp?ID=373 (cited Dec. 2006).

[18] J.A. Jones, M.J. Harrold, J. Stasko, "Visualization of Test Information to Assist Fault Localization", *Proc. Int'l. Conf. Software Eng. (ICSE'02)*, 2002, pp. 467-477.

[19] S. Jungmayr, "Identifying Test-Critical Dependencies", *Proc. 18th IEEE Int'l. Conf. Software Maintenance (ICSM'04)*, 2004.

[20] M. Lindström, "Ensuring Availability and Access to New and Existing Technologies in Cellular Terminal Business", *Ph.D. Dissertation. Helsinki University of Technology (HUT)*, 2003, ISBN 951-22-6521-4.

[21] G. Mishegri and Z. Su, "HDD: Hierarchical Delta Debugging", *Proc. 28th Int'l. Conf. Software Eng. (ICSE'06)*, 2006.

[22] S. Mouchawrab, L.C. Briand and Y. Labiche, "A measurement framework for object-oriented software testability", *Journal of Information and Software Technology 47*, 2005, pp. 979-999.

[23] G. Myers, T. Badgett, T. Thomass and C. Sandler, *The Art of Software Testing*, John Wiley & Sons, Inc, 2004, 256 pp.

[24] R.E. Park, "Software Size Measurement: A Framework for Counting Source Statements", *SEI Technical Report CMU/SEI-92-TR-25*, Sept.1992.

[25] H. Stork, "Structuring Process and Design for Future Mobile Communication Devices", http://videos.dac.com/43rd/slides/stork.pdf (cited Dec. 2006).

[26] Texas Instruments, "OMAP 2 Architecture: OMAP2420 Processor", *whitepaper*, 2005, http://focus.ti.com/pdfs/wtbu/TI_OMAP2420.pdf (cited Dec. 2006).

[27] Texas Instruments, "SmartReflex power and performance management technologies", *whitepaper*, http://focus.ti.com/pdfs/wtbu/smartreflex_whitepaper.pdf (cited Dec. 2006).

[28] T. Vaskivuo, "Correlation Methods for Analysis and Visualisation of Software Run-time Behaviour", *Proc. 9th Int'l. Conf. Software Eng. and Applications 2005 (SEA 2005)*. Phoenix, Arizona, 14 - 16 Nov., 2005.

[29] J.M. Voas and K.W. Miller, "Software Testability: The New Verification", *IEEE Software*, May 1995, pp. 17-28.

[30] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input", *IEEE Trans. Software Eng.*, vol. 28, no. 2, Feb. 2002.

11th European Conference on Software Maintenance and Reengineering (CSMR'07)
0-7695-2802-3/07 $20.00 © 2007 IEEE

PAPER II

Towards Trace Based Model Synthesis for Program Understanding and Test Automation

In: Proceedings of the 2nd International Conference on Software Engineering Advances, ICSEA 2007, Cap Esterel, French Riviera, France, 25–31 August, 2007.
10 p. © 2007 IEEE.
Reprinted with permission from the publisher.

Towards Trace Based Model Synthesis for Program Understanding and Test Automation

Teemu Kanstrén
VTT Technical Research Centre of Finland,
P.O.Box 1100, FI-90571 Oulu, Finland
teemu.kanstren@vtt.fi

Abstract

Effective maintenance and evolution of complex, software intensive systems requires understanding how the system works and having tests available to verify the effects of changes. Understanding complex systems is difficult, and testability of these systems is often low due to design constraints, system complexity and long-term evolution. Thus understanding the system and adding new tests is difficult. Yet, especially in these cases, the understanding and tests are important to verify the system correctness over long-term evolution. This paper discusses synthesizing models from system traces and using these models to facilitate program understanding and test automation. Basing the models on execution traces allows generation of automated tests even for low testability systems. Generating and visualizing abstracted models facilitates program understanding, which helps in system maintenance.

1. Introduction

In the course of system maintenance and evolution, existing functionality is changed and new functionality is added. Making changes safely to an existing system requires both an understanding of the system and the availability of test cases to verify the effects of the changes on the system. However, many times, especially with legacy systems, a good understanding and related tests cases are not available. Thus, adding new tests for verifying the system behavior and the effects of the changes are needed. However, without a good understanding of the system and its support for testability this is problematic.

Understanding a complex system is a difficult task. Even with understanding, the test implementation can still be challenging. For example, in large legacy systems, the code base can grow to millions of lines of code and hundreds of components, the effects of change can ripple over large parts of the system, and the same code needs to work on different execution platforms [15]. Combined with a constantly evolving underlying platform, the understanding needs to be kept current and regression tests are needed to verify the functionality.

Yet it is often the case, especially with legacy systems, that the functionality to test can not be easily separated from the system or accessing the required internal information is not possible. In complex systems with long-term evolution, the testability of the system is often low, and different features become tightly integrated into the system. The system platform, such as an embedded real-time system, can further set more constraints. In these cases, forcing a given control path and observing system internal states in a test case can be problematic, which makes applying traditional ways of building test cases problematic [15].

This paper addresses these issues by presenting a technique based on using models synthesized from traces of the system under test (SUT) execution, targeting especially systems with deeply embedded and complex functionality. By producing an abstracted model of the SUT execution, the technique facilitates program understanding. As the technique is based on system traces, it can be applied without specific testability support from the SUT architecture and design. The generated model is used as a reference (describing what is expected), against which traces from the actual current implementation are verified in regression testing. To demonstrate the application of the technique in practice, it is applied on a messaging middleware component.

This paper is structured as follows: Section 2 discusses the basic concepts. Section 3 discusses synthesizing models from the system traces and using them as test cases. In section 4 the technique is demonstrated by applying it on a real world system. Section 5 discusses the technique. Section 6 discusses related work. Finally section 7 concludes the paper.

International Conference on Software Engineering Advances(ICSEA 2007)
0-7695-2937-2/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

2. System Tracing

System tracing is the basic concept of gathering information about the dynamic behaviour of the system. This can be done at multiple levels as illustrated in Figure 1. The left column in the figure lists the different tracing instruments. The middle column shows the location of tracing instrument in the system. The right column describes the context dependency of the different tracing instruments. Techniques with higher context dependency, such as logging statements, require more manual work but also best capture the specific functionality of a system. The techniques with lower context dependency, such as platform instruments, can be better automated over different system and platforms, but capture only an overview of the system. Thus the use of different levels of tracing and their combinations is a trade-off between how much can be automated and how specific information is needed.

Instrument	Location		Context dependency
1. Logging statements	Application code	MANUAL	High
2. Instrumented methods	Application code	AUTOMATED	
3. Instrumented system operations	System libraries		
4. Instrumented OS services	OS Kernel		
5. Sampled program execution	OS Kernel / Platform services	PLATFORM	
6. (sampled) Platform metrics	Platform services		
7. Platform instruments	Platform services		Low

Figure 1. Trace levels (adapted from [26]).

2.1 Tracing Levels

Logging statements (trace level 1) are added to the source code to trace information about system states and behaviour at different points of execution. They are the most context dependent traces, as all the traces need to be added on individual basis and are different for each application. Similarly, by *method instrumentation* (trace level 2) system call graphs and method parameters can be acquired. Collection of this data can be automated with tools such as AspectJ [1] and GNU profiler [9], but it is context dependent in the sense that a call graph is very specific to a system.

System operations as well as *operating system (OS) services* are shared across many applications and can be used to provide automated instrumentation of system execution. While each application will have a unique trace, it will be within the context of the functionality provided by the libraries and system services. For example, middleware communication data (level 3) or information on task and thread context switching [13] (level 4) can be traced. This data can be considered common high-level functionality of applications.

The implementation of platform level trace instruments (trace levels 5-7) depends on the execution environment (the platform). In embedded systems the platform is the hardware on which the embedded software runs. In modern application environments it is the virtual machine (VM) on which the code is executed. Here the focus is on embedded systems as a platform, although modern VM environments also provide good support for platform level tracing, such as the Java Platform Debugging Architecture (JPDA) [14].

Sampling program execution provides abstraction over system behaviour. By sampling program execution at fixed intervals, it is possible to get a statistical view of the behaviour. For example, in embedded systems such sampling can be implemented using timer interrupts to record program counter values [25]. *Platform metrics,* such as CPU load and memory allocations, can be collected in a similar way by sampling, or by using triggers in system state to record the metric values. In embedded systems, these triggers could be for example interrupts. *Platform instruments* include counters and similar information provided by the platform internal implementation. In embedded systems these can be hardware counters. In VM environments, platform level tracing typically requires supporting hooks from the VM.

In tracing binary executables, such as COTS components without source code, it is possible to use any of the instruments on levels 3-7, but the possibilities at level 1 and 2 are limited. For level 1, possibilities are limited to instrumenting the glue code that integrates the components together. For level 2, bytecode instrumentation in VM environments can be used, but the code is likely to be obfuscated and of limited use as the code structure is unknown. In embedded systems, platform supported instruments, such as using level 5 sampling need to be used.

On the lower levels of 3-7, instrumentation is dependent on the options provided by the platform on which the code runs. At level 3, if there is no access to system library code and the libraries do not provide trace interfaces, possibilities are limited to what can be traced for example from the glue code. Levels 4 and 5 require access to, or support from the OS kernel. Levels 5-7 also require access or support from the execu-

International Conference on Software Engineering Advances(ICSEA 2007)
0-7695-2937-2/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

tion platform. Thus the available tracing options are set by the context in which tracing is done.

2.2 Trace Based Program Understanding and Test Automation

In this paper, the goal in considering traces as a basis for testing is two-fold; understanding the system, and having test cases available to document and verify this understanding and the important properties of the system behaviour across project evolution. Modern software intensive systems are complex and their behaviour is difficult to understand. Thus, in testing an understanding for the system needs to be built, and assumptions about its behaviour need to be validated. Similar to previous work [8], this paper considers tracing actual system execution to be the best source of information for actual system behaviour. Once this understanding has been built, it is preserved in verifiable form as testable models. By building the models to be easily evolved, they can also be used to validate future changes and the validity of our understanding of the effects of these changes on the system.

From this point of view, there is a need to consider the different levels of tracing and how they can be utilized in automated support to achieve these goals. This paper counts two types of testing that can make use of this type of automation; *functional testing* which verifies formal properties of system execution, and *statistical testing,* such as performance testing, which is interested in verifying certain statistical properties and constraints of system execution, such as response times and throughput. This paper mostly considers functional testing but also discusses how the same techniques can be applied to statistical testing.

From the *functional* testing perspective, system functionality is unique and thus context dependent. From this viewpoint, the best tracing level is adding level 1 logging statements to the system to collect the information of interest. This type of instrumentation makes it possible to define the traces at the conceptual level, to best describe the system execution in a way that facilitates program understanding. However, this has the drawback of requiring the most resources and human work for instrumentation. From this viewpoint, the less context dependent traces need to be considered as their use can be more automated and requires less human resources. Thus, when using this technique, the use of different tracing levels in synthesis, and their trade-offs need to be considered.

From the *statistical* testing point of view, the least context dependent tracing levels provide equally useful data. This is especially true, if the platform level support is considered during design time such as in designing embedded systems where hardware and software co-design is considered. However, in most of these cases, also the more context dependent trace mechanisms need to be used to collect relevant data from system execution. Only a limited amount of tracing can be supported at platform level and trace points in software code are needed to collect information on specific properties of the system. Thus statistical type of testing can make better use of the full range of tracing instrument techniques. However, this type of testing is left out of the scope of this paper.

3. Model Synthesis and Test Generation

The approach taken to using system traces as a basis for program understanding and test automation is the following. The tracing instruments described in Figure 1 are used to gather the trace. These traces are analysed to gain an understanding of the SUT. From the set of gathered trace elements, the ones most important for the functionality at hand are selected, they are refined to describe their relations and constraints to formalize a verifiable model. The resulting model is called the synthesized trace based model of the system. To document the understanding of the system, the model is kept as a regression test. Current and future system behaviour is verified by comparing the system execution against this model.

3.1 Abstraction Levels

As discussed in [21], software can be modeled at different abstraction levels. To synthesize models for a system, the abstraction level of the models needs to be defined. A common concept in reverse engineering (RE) techniques is using data at the detail level of method calls to model the behavior of a system [11]. Traces such as these, describing the detailed low-level execution of a system could be used as regression tests to verify that the SUT behaviour is not changed. However, from the viewpoint of this paper this approach is problematic; the amount of trace generated quickly grows very large and test cases built on such traces are difficult to understand and fragile.

For program understanding, large traces with excess detail are not optimal. Also, although this type of tracing has been used for program understanding [6], it is dependent on factors such as method naming conventions, which are prone to change and not uniform across projects. From test generation perspective, another problem in using detailed data such as call-graphs as a basis for test cases, is that it will cause failures at the smallest changes in the system. For exam-

International Conference on Software Engineering Advances(ICSEA 2007)
0-7695-2937-2/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

ple, simple refactorings may not be significant to the functionality under test but can change the call-graph.

However, in the case of a failing test case it can be useful to have more detailed information available for locating the cause of the failure. Thus, while this paper mostly considers higher level abstractions, it is also seen as useful to maintain traceability information from the abstracted models to the actual source code, similar to [3].

3.2 Model Elements

In this paper, a test case is defined as a sequence of steps, each tied to one or more events and defining their relations to other events and attributes. Table 1 lists the generic model elements defined. Events are defined in similar way as [3]: Meaningful properties and actions in program execution. These can be at different abstraction levels, such as sensor input, message send or closing a file. However, they should represent relevant concepts to the program execution and the verification of the functionality under test. Attributes are associated with events and can represent any relevant data for the event such as timing, message content, object identifiers and data values.

Table 1. Generic model elements.

Element	Description
Test start	Test case execution starts.
Test end	Test case execution ends.
Test input	Test case input data.
Test output	Test case output data.
Precedence	One event must occur before another.
Duration	Event time interval.
Task	Sequence of related events.
Synchronity	Synchronization between tasks.
Messaging	Communication between tasks.
Match	Test sequence step event or attribute must match that of another step.
Alternate	When only a subset of a number of given events or attributes is required.
Inclusion	One event must include another.
Exclusion	One event excludes another.
Reference	References another test sequence step.
Repetition	Repeating loops in a model.

Test start and *end* are useful synchronization points for different communicating tasks. *Input* and *output* are basic properties needed in software testing. They can be related to a test case as a whole or to smaller subtasks inside a test case. For test sequence ordering, two levels of *precedence* are defined: one event occurs *immediately* after another event, or as any event in *temporal* order after another. *Event duration* is the time an event is active. Not all events have a meaningful duration, for example receiving a message can be considered a singular event without a specific duration. On the other hand another event, such as processing the message, can have a more meaningful duration.

Tasks group a set of related events together to form a sequence of events realizing a higher-level concept. This can be realized as events of a single thread but more generally they are any events that form a higher level concept together, which can be spread across several threads or occur interleaved with other concepts. Interaction related elements include *synchronization* and *messaging*, which are some of the basic properties of communicating and concurrent systems. These include such properties as points of mutual exclusion and process communication.

Single steps of system execution are rarely meaningful alone, and this paper uses a set of elements to describe their relations to each other. Events and attributes can be required to be *matching* the attribute values of other events and attributes, *alternative* to each other, *including* (requiring) or *excluding* (disallowing) another event or attribute. *References* to other test steps are needed to describe these relations. Combinations of the model elements are also possible, such as one of several events is allowed (*alternative*) but not many of them (*exclusion*). *Repetition* is needed to express loops, which are basic concepts in software implementation. Not all trace events are relevant to a feature, and as such only the relevant ones are included in the model and the rest are ignored.

3.3 Model Building

The process of building models from the system traces is shown in Figure 2. To build a model of the system behavior, the feature of interest is first executed to gather a trace. To gather the trace the system is instrumented for tracing by using any combination of the techniques as discussed in section 2.1. The system feature is executed to produce the trace and store the produced trace as a basis for our model. To formalize the relations of the trace steps as a model of the systems expected behaviour, the trace is refined with the model elements described in Table 1. This produces the model that is used to document the understanding of the system and as a basis for regression testing.

Step 2 in Figure 2 describes executing system functionality. This refers to any means to execute the functionality but is best implemented as with automation, exercising the code in repeatable fashion. This way the same method of exercising the code can also be applied for the regression testing. However, other means such as manually exercising the program can also be used if necessary.

Finally, as understanding for the system will likely grow over time, and as the system itself evolves and

changes, the models need to be refined. This requires incorporating new trace elements into the model and possibly adding new trace instrumentation to the system. The model updating can be automated to the extent that a new trace is collected and the model is updated with selected elements from the new trace. Only the refining model elements from Table 1 need to be added to the new trace elements and the old elements that are related to the new elements need to be updated for this part. The need to update the model can also come from for example finding faults in functionality that should be covered by regression tests based on the models, or if the model is otherwise found not to correctly express all the necessary information for the system.

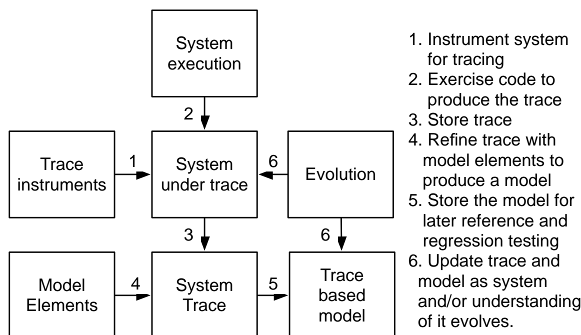


Figure 2. Model building.

3.4 Regression Testing

The process of using the generated models for regression testing is shown in Figure 3. System execution in the figure is again similar to what is described in section 3.3. The previously generated model is used as a basis for regression testing. This model is used to describe what is expected of the SUT. To verify the actual system behavior against this expected model, the actual execution of the SUT is traced and the trace is checked against the model for the expected. Analysis of the test results determines if the test was successfully passed. A failure can be analyzed and the model can help in locating the failure by highlighting the parts of the models and traces that do not match.

The algorithm for checking the trace against the model is described in Figure 4, This consists of checking if the trace elements meet all the required event and attribute values, as well as whether the trace fits within the constraints set by the expected model. The same instrumentation that is used in generating the expected model can be used to trace the system for regression testing. Test results are stored in a test log, which can be visualized for easier analysis as shown in Figure 6.

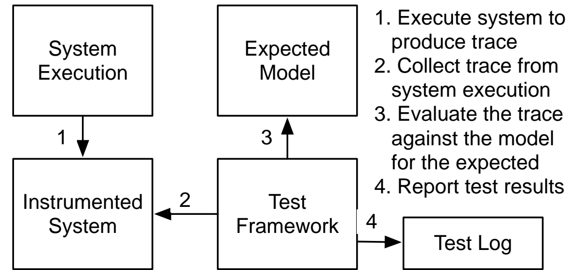


Figure 3. Regression testing.

```
-For each model step
  -Check if the step is found in the trace
  -If the step is not found
    -Mark the step failed, showing the given step as
     missing in test log
  -If the step is found
    -Check all required properties set for the step as
     specified by the model elements in Table 1.
    -If the require properties are not met
      -Mark the case as failed, with the reason of
       failure shown in the test log
-If all steps passed the check mark the test as passed
-Else mark the test as failed
```

Figure 4. Trace checking algorithm.

4. Example Case

4.1 Generic Communication Middleware

To demonstrate and validate the practical usefulness of the approach, it is applied on a system called Generic Communication Middleware (GCM). GCM is a middleware for application messaging in heterogeneous distributed computing environments [20]. It is targeted to facilitate the development of distributed applications into heterogeneous computing environments, including devices with limited resources. For demonstration, the basic middleware feature of sending a message is considered.

In tracing GCM, manual logging statements (level 1 as defined in Figure 1) are used. GCM does not use third party components, or have other external dependencies, and the source code is available, which makes it possible to instrument all parts of the system. It has multithreaded functionality, but to illustrate the technique is a clear and simple way, this is not the focused on here. The aim in the trace definition has been to craft it to be expressive, making the demonstration easier to understand.

As the first step, trace statements are added to the code. The trace is iteratively refined by executing the system, observing the trace and evaluating its expres-

siveness in describing the system functionality. The resulting trace for the feature is shown in Figure 5 on the right hand side, which also lists the three separate sequences visualized on the left hand side. These separate sequences illustrate the grouping of subsequences as *tasks* (as in Table 1). The sequences are initialization, sending normal message and sending control message. As shown, sending normal message is a sequence interleaved with other tasks, while the two others are continuous.

These sequences describe the basic functionality of sending a message with GCM. To send a message, the GCM must first be initialized. Once initialization is done, messages can be sent. However, before the first message is sent, a control message must be created and sent to establish a connection between the GCM client and server. Sending a control message is done automatically by the GCM service when the first message is sent. This whole functionality is shown in the three separate sequences in Figure 5. When the initial trace is first collected, the sequences are of course not visible and the trace is a continuous sequence of events. This is why the trace needs to be refined to generate a usable model to serve as a document for program understanding and a verifiable test case for future executions.

4.2 Generating a Model and a Test Case

Armed with the trace, it is possible to generate models from it and to use these as regression tests. Building the model includes parsing the events and attributes from the trace data and associating them together. This part can be fully automated as long as a formal trace format is used, as is already done in Figure 5.

For generating models, these traces can be viewed from different viewpoints depending what is being modeled. One option is to build one large model to include the whole trace sequence at once. However, to keep the models focused and easy to handle, they have been partitioned to separate models. This gives an understanding of the individual functionality and the smaller models can later be mapped together to describe the larger functionality.

Each of the sequences in Figure 5 could be defined as a model. However, here an example is used that is a subsequence of both sending a normal message and sending a control message, and illustrates a real issue faced in the actual development of GCM. This is illustrated in Figure 6, which shows a model for writing the actual message data (sending it across the network) from the client to the server. The *expected* column describes the initial model, and the textual description of the steps describes the *expectations* for each step and their relations to other steps. The steps and textual descriptions are mapped by their number id values. The *actual* column describes a trace from the actual implementation and is discussed in the next subsection. Table 2 maps the notation elements to the model elements from Table 1.

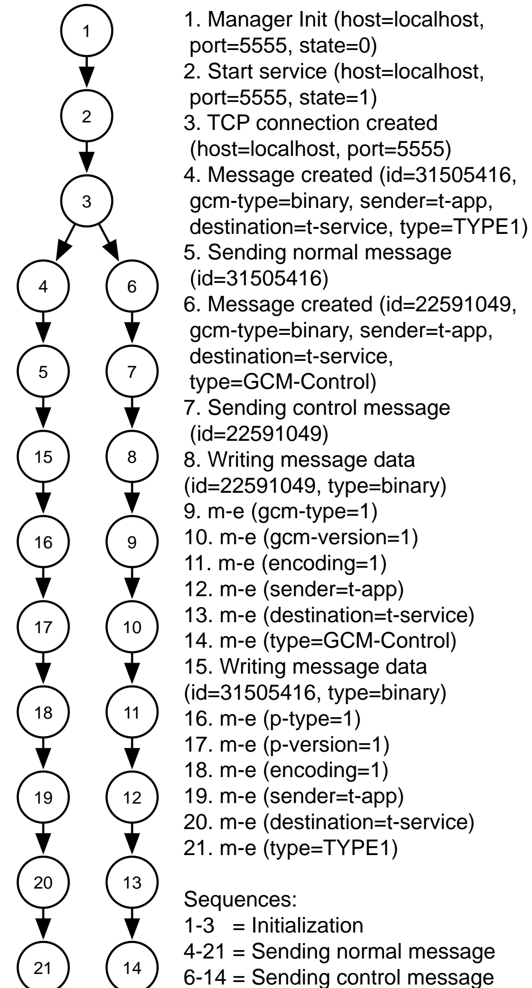


Figure 5. GCM Trace.

In GCM's evolution, the binary protocol for passing the message parameters has been changed from passing of parameter length (step 7 in Figure 6) and value (step 8) to include the parameter type (step 6) as well. At this time the protocol version has also been changed to 2. The previous functionality of passing the parameters produced a trace similar to the expected trace in Figure 6, without step 6. From the GCM version 1 trace the *expected* trace shown in the figure was generated by simply changing the expected protocol type to 2 and adding the parameter type trace element (step 6) to the model.

In the actual implementation of this change, this type of a test case was not available. Instead, the im-

plementation was debugged using a network monitor to trace and examine client-server communication. The updated version was found not to work properly and the problem was traced to faulty implementation of the parameter type. The fault was introduced in adding the new type field to the parameters. It was added as the last element in the parameter data when it should have been the first.

Table 2. Test notation.

Notation	Model Element
<N>	Reference to another step number N.
attr	Matching attribute.
event=	Matching event.
after=	Temporal order.
first-after=	Immediate order.

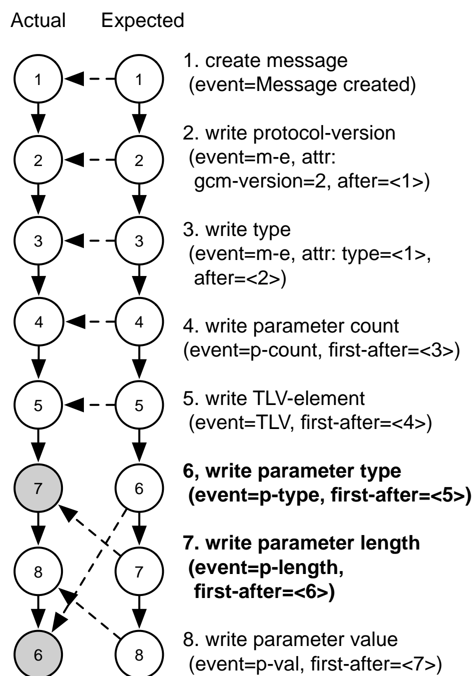


Figure 6. A model and its verification.

4.3 Test Execution and Analysis

Figure 6 also describes the verification of the GCM trace against the expected model. As described earlier, the actual column shows the trace for the actual execution of the system and the expected column along with the textual description describes the generated model for the system. The dashed arrows show how the steps in the expected model are matched to the trace elements of the actual execution. For example, the expected trace element 6 (writing parameter type) is matched to element 8 in the actual trace. The highlighted (bold) trace elements and steps show failing steps in the trace validation.

In the figure, the last three steps are the ones of interest. Steps 6 and 7 fail as their event relations (*first-after* tags) do not match the actual trace. Step 8, though seemingly misplaced in the trace, does not fail as its event relations match the actual trace: it is the first event to happen after the event of step 7. The parameter type event (step 6) should happen as the first event after step 5, but it is the third element and thus this step fails. Similarly, step 7 should happen as the first event after step 6 but it happens two events before. Thus, the two failing steps and their associated trace elements effectively highlight the failure in the test case and its cause. This shows how this type of testing can effectively find the failure and its cause.

5. Discussion

The intent for this paper was to demonstrate the described technique by generating models for the GCM implementation and using these as regression tests. While there were no expectations where this would lead, the visualization and analysis of the traces and models helped understand the system and highlighted areas in need of improvement. First, the message objects created by the user include the information identifying the binary protocol and its properties such as encodings. The user message objects should only contain the message contents, not protocol level details.

Second, the implementation also adds unnecessary details to the control message. The control message contains data copied for each field in the message from the normal message that is being sent in the system. As GCM is targeted to address the needs of constrained embedded devices, this consumes unnecessary resources and should be eliminated. In the GCM specification, this part is not described - only the type of the message is defined. Thus, this also highlights a possible need for refinement in the specification.

With the exception of instrumenting the system (step 1 in Figure 2) and model refinement (step 4 in Figure 2), all steps in applying the technique can be fully automated. When the less context dependent trace mechanisms described in Figure 1 can be applied, parts of step 1 can also be automated. Different parts of task 4 can also be automated. For example, promising techniques exist that can be applied to identify tasks from system execution [5]. In general, step 4 is an interesting future research topic for further automation. This could include rule based approaches, learning algorithms, and other similar algorithms. Model refinement can also be made easier by providing visualizations such as described in Figure 6 and making it possible to

IEEE COMPUTER SOCIETY

do the refinement through visual manipulations as well as with automated assistance.

While it is possible to apply the technique presented in this paper to all types of functional testing, it is not always cost-effective. The best application domain is in systems where the functionality is highly embedded in the system and can be expressed as sequences of events. This can be traditional HW/SW embedded systems but also any type of system that is sufficiently large and complex, and needs to be tested as a larger integrated product. This includes cases, where the system behaviour can not be effectively tested from external interfaces, when the behaviour is complex, and needs to be understood and debugged with regards to its inner working. In short, this means low testability systems and features, where building a full set of external observability features is not feasible.

For constrained systems such as real-time systems, the extra trace overhead (probe-effect) can also make use of this type of technique more challenging. Using and combining traces from separate parts of distributed system also needs further consideration. Thus future work needs to address the limitations of the technique and how it can be most cost-effectively applied in different contexts. In general, these constraints as well as other automation aspects of the technique need to be addressed with tool support.

6. Related Work

This section reviews related work. Model based testing (MBT) is mostly related to modeling what is expected of the SUT based on its requirements and using these models to generate test cases. Reverse engineering (RE) addresses the other side of the equation: what is the actual implementation provided by the SUT. Testing tools and techniques that make use of trace based models are also considered. As it is possible to use the same concepts from these techniques to model a system for testing such as in this paper, an overview is given on how they trace and model the systems as well as how the models are used.

In MBT, the SUT is modeled based on its requirements and these models are used to generate test cases for the SUT. The basic idea is to explicitly describe the requirements as models and verify that these models accurately reflect what is expected of the system [21]. Once verified, these models and traces derived from them are used as a basis for generating test cases for the SUT. The models take different forms depending on what is being modeled and how. Different types of models used include data models [7] [23], behavioral models [1] [21] [22] and domain specific models [16] [17] [19]. The models can be described at different levels of abstraction for different viewpoints [21].

In general, MBT tools typically exercise the system as a black box through external interfaces that need to be supported by the system design [24]. Thus they focus mostly on well defined inputs and outputs of a system, whereas the technique in this paper is more applicable to the inner workings of low testability systems. As there are common properties in both, such as using models and observing the system, possible synergies may exist. These are however, out of the scope of this paper.

Bertolino et al. [4] also describe what they call anti-model based testing. They focus on creating a set of traditional test cases for black-box component based software and use these test cases to gather traces for the system. From these traces they try to synthesize a behavioral model for the system. They describe their model as a state-machine, and their tracing mechanism as instrumenting component glue code. Thus the approach in this paper considers more specific application of models for program understanding and regression testing, as well as a wider range of tracing techniques.

Many RE tools apply different abstractions and filterings on the traces to limit the amount of data to be processed, and use the processed data to provide models and visualizations of the systems [11]. Examples include discovering and visualizing patterns in the traces, generating sequence and scenario diagrams, graphs and custom visualizations. The main difference with this paper is that RE is interested in generating various models of the system and stops there, whereas the technique presented in this paper also applies these models for testing.

Huselius and Andersson [13] insert probes into the system to monitor context switches and system calls. They use context switches differentiate tasks that execute jobs and use these concepts as basis for their modeling. They also discuss inserting data probes to monitor selected variables within the system and to represent the system state. Thus, their data set includes the context switches, selected system calls, and selected state variables. One of the uses they describe for their technique is the validation of COTS components, though they do not elaborate on it further.

Lam and Barber [18] consider modeling agent software for the purposes of human comprehension. They use use specific agent concepts for their modeling. Modeling the software starts by defining the agent concepts in the source code as logging statements that gather data related to these concepts. As the software is executed, this information is logged and the model is refined with these observations. The models are stored in a knowledge base. Enough information is logged for

each event to associate it with its exact place in the source code. Based on the different agent concepts and their relations, they build an overall model that is intended to help in understanding the agent system.

Ducasse et al. [8] use logic based queries of the SUT execution traces to test legacy systems. Their traces include events and object states, recorded from program execution. Events are messages between objects, including parameter and return values. To validate assumptions about the SUT, they use logic queries on the traces and define a set of trace-based logic testing patterns. They use these tests to validate that legacy systems remain the same after changes and to help understand a program by creating and validating assumptions about it. Their work is closely related to this paper: both use traces to test and facilitate understanding of legacy systems. They, however, use logic queries to assert the trace data, whereas this paper uses visualizable traces and models. This paper also bases the models on higher level abstractions and formalized event relations, whereas they use more detailed traces at the level of method calls.

TextTest is a tool to create regression tests from log files [1]. A log file is stored and set as the "`standard" against which further test executions are compared. By text comparison, it is determined whether the test passes or fails. A failing test case is shown with the differences highlighted in the text files. TextTest provides an opportunity to use regular expressions to define lines of text that will be excluded from the comparison. TextTest, uses traces for testing in a similar way to this paper, but focuses on the manual trace level and on directly matching the trace file, whereas this paper uses a model based on broader levels of traces, events and their relations. In addition, the technique presented in this paper also aims to support system understanding.

Model checking is a process of formally checking a model of the SUT in relation to a set of specified properties, such as deadlocks and user assertion failures [10] [12] [22]. Model checking tools use two different approaches: check models derived and abstracted from the source code, or drive the execution of the system and use it to represent the state-space [10]. Checking the models is performed using algorithms that explore their state-space for the desired properties. The states consist of process interactions and similar properties of the system [10]. While this paper is also interested in generating and checking a model of the actual execution against its expected model, no state-exploration is performed but rather conceptual matching of the two models is used.

7. Conclusions and Future Work

This paper presented a technique for trace based model synthesis for program understanding and test automation. Its application was demonstrated with a middleware component. It was shown how trace based models for existing systems can be generated and evolved, and how they are useful in program understanding. It was also shown how the models can be used as a basis for regression testing, which can be effective in finding failures and locating their causes. As the technique only requires having traces of system execution, it is also applicable to many low testability systems such as constrained embedded systems and legacy code.

Considering automating the process of using the technique, it was shown how most of the technique can be fully automated. While currently no automated tool support exists, in the future to effectively use the technique, automated and integrated tool support is needed to trace systems, refine them to models, execute them as regression tests and to produce visualizations from the test logs. In this regard, the parts still needing the most manual effort and thus most potential for future research are automated trace instrumentation and model refinement from the trace(s). Further validating and refining the technique in different systems and environments is also needed.

8. Acknowledgements

The author wishes to thank the anonymous referees for their constructive comments. The publication of this paper has been supported by the ITEA TWINS project, and the work of the author has also been supported by the Jenny and Antti Wihuri Foundation.

9. References

[1] J. Anderson, and G. Bache. "The video store revisited yet again: Adventures in gui acceptance testing", *Proc. Extreme Programming and Agile Processes in Software Eng.*, pp. 1–10, 2004.

[2] AspectJ, http://www.aspectj.org. [referenced May-2007]

[3] M. Auguston, J. B. Michael, and M.-T. Shing, "Environment behavior models for scenario generation and testing automation", *Proc. 1st int'l. workshop on Advances in model-based testing (A-MOST'05)*, pp. 1–6, New York, USA, 2005. ACM Press.

[4] A. Bertolino, A. Polini, P. Inverardi, and H. Muccini, "Towards Anti-Model-Based Testing", *Fast Abstracts in Int'l. Conf. Dependable Systems and Networks (DSN 2004)*, Florence, 2004.

[5] J. E. Cook, and Z. Du, "Discovering thread interactions in a concurrent system", *J. Syst. Softw.*, 77(3):285–297, 2005.

[6] B. Cornelissen, A. Deursen, L. Moonen, and A. Zaidman, "Visualizing Testsuites to Aid in Software Understanding", *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR'07)*, 2007.

[7] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice", *Proc. 21st Int'l. Conf. on Softw. Eng. (ICSE'99)*, pp. 285–294, Los Alamitos, CA, USA, 1999.

[8] S. Ducasse, T. Girba, and R. Wuyts. "Object-oriented legacy system trace-based logic testing", *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR'06)*, pp. 37–46, 2006.

[9] GNU Profiler, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html. [referenced May-2007].

[10] P. Godefroid, "Software model checking: The verisoft approach", *Form. Methods Syst. Des.*, 26(2):77–101, 2005.

[11] A. Hamou-Lhadj, and T. C. Lethbridge, "A survey of trace exploration tools and techniques", *Proc. conf. of the Centre for Advanced Studies on Collaborative research (CASCON '04)*, pp. 42–55, 2004.

[12] G. J. Holzmann, "The model checker SPIN", *IEEE Trans. Softw. Eng.*, 23(5), pp.279–295, May 1997.

[13] J. Huselius and J. Andersson, "Model synthesis for real-time systems", *Proc. Conf. Softw. Maintenance and Reengineering (CSMR'05)*, pp. 52–60, Mar. 2005.

[14] Java Platform Debugging Architecture, http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html. [referenced May-2007]

[15] T. Kanstren, M. Hongisto, and K. Kolehmainen, "Integrating and testing a system-wide feature in a legacy system: An experience report", *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR'07)*, 2007.

[16] M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, and M. Satama, "Towards deploying model-based testing with a domain-specific modeling approach", *Proc. Testing: Academic & Industrial Conf. on Practice And Research Techniques (TAIC-PART '06)*, pp. 81–89, 2006.

[17] G.-B. Kim, "A method of generating massive virtual clients and model-based performance test", *Proc. 5th Int'l. Conference on Quality Software (QSIC'05)*, pp. 250–254, 2005.

[18] D. N. Lam, and K. S. Barber, "Comprehending agent software", *Proc. 4th Int'l. joint conference on Autonomous Agents and MultiAgent Systems (AAMAS'05)*, pp. 586–593, 2005.

[19] E. M. Olimpiew, and H. Gomaa, "Model-based testing for applications derived from software product lines", *Proc. 1st int'l. workshop on Advances in model-based testing (A-MOST'05)*, pp. 1–7, 2005.

[20] D. Pakkala, P. Pakkonen, and M. Sihvonen, "A generic communication middleware architecture for distributed application and service messaging", *Proc. Join Int'l. Conf. on Autonomic and Autonomous Systems and Int'l. Conf. on Networking and Services (ICAS-ICNS 2005)*, Oct. 2005.

[21] A. Pretschner, W. Prenninger, S. Wagner, C. K uhnel, M. Baumgartner, B. Sostawa, R. Zolch, and T. Stauner, "One evaluation of model-based testing and its automation", *Proc. 27th Int'l. Conf. on Softw. Eng. (ICSE'05)*, pp. 392–401, 2005.

[22] Robby, M. B. Dwyer, and J. Hatcliff, "Bogor: an extensible and highly-modular software model checking framework", *Proc. 9th European Softw. Eng. Conf. held jointly with 11th ACM SIGSOFT Int'l. Symposium on Foundations of Softw. Eng. (ESEC/FSE-11)*, pp. 267–276, 2003.

[23] P. Schroeder, E. Kim, J. Arshem, and P. Bolaki, "Combining behavior and data modeling in automated test case generation", *Proc. 3rd Int'l. Conf. Quality Softw. (QSIC '03)*, pp. 247–254, Nov. 2003.

[24] Utting M., Legeard B., *Practical Model-Based Testing - A Tools Approach*, Morgan Kaufmann, 456 pages, 2007.

[25] T. Vaskivuo, "Correlation methods for visualization of software run-time behaviour", *Proc. Softw. Eng. And Applications (SEA 2005)*, Nov. 2005.

[26] T. Vaskivuo, "Correlation methods for visualization of software run-time behaviour", *Presentation at Softw. Eng. And Applications (SEA 2005)*, Nov. 2005.

PAPER III

Towards a Deeper Understanding of Test Coverage

In: Journal of Software Maintenance and Evolution: Research and Practice, JSME, Vol. 20, No. 1, 2008. Pp. 59–76.
Reprinted with permission from the publisher.

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **20**:59–76
Published online 5 November 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.362

Research

Towards a deeper understanding of test coverage

Teemu Kanstrén*, †

VTT Technical Research Center of Finland, P.O. Box 1100, Kaitoväylä 1, Oulu FI-90571, Finland

SUMMARY

Test coverage is traditionally considered as how much of the code is covered by the test suite in whole. However, test suites typically contain different types of tests with different roles, such as unit tests, integration tests and functional tests. As traditional measures of test coverage make no distinction between the different types of tests, the overall view of test coverage is limited to what is covered by the tests in general. This paper proposes a quantitative way to measure the test coverage of the different parts of the software at different testing levels. It is also shown how this information can be used in software maintenance and development to further evolve the test suite and the system under test. The technique is applied to an open-source project to show its application in practice. Copyright © 2007 John Wiley & Sons, Ltd.

Received 3 July 2006; Revised 18 September 2007; Accepted 19 September 2007

KEY WORDS: test granularity; level of testing; test optimization; test coverage

1. INTRODUCTION

Test suites typically contain different types of tests such as unit tests, integration tests and system tests. In practice this means that test suites consist of test cases that exercise the system under test (SUT) at varying granularities. Some exercise smaller parts of the SUT at a finer granularity, while others exercise larger parts at a coarser granularity. The spread and ratio of these different types of tests vary for different test suites and different parts of the SUT. Some suites may contain fewer tests that exercise larger parts of the SUT and some suites contain more tests that exercise smaller parts of the SUT. In both software development and maintenance, different types of tests have different benefits and roles, such as verifying the functionality of individual components, confirming their interactions and aiding in debugging.

*Correspondence to: Teemu Kanstrén, VTT Technical Research Center of Finland, P.O. Box 1100, Kaitoväylä 1, Oulu FI-90571, Finland.
†E-mail: teemu.kanstren@vtt.fi

Copyright © 2007 John Wiley & Sons, Ltd.

WILEY InterScience® DISCOVER SOMETHING GREAT

Traditional measures of test coverage focus on measuring how much of the total SUT has been exercised by the test suite. Various coverage measures include measures such as statement, path and decision coverage [1]. These traditional types of code coverage are useful for seeing which parts are not yet under test and for getting an overview of how much of the SUT is tested in general. However, once previously uncovered code is brought under test, traditional code coverage measures only tell us that the code is covered by some test in the test suite. These measures do not tell anything about the types of tests exercising the piece of code. Thus, if we want to understand better how the SUT is covered by different types of tests, the traditional view of test coverage does not provide a good view for this purpose.

This paper proposes a way to get a deeper understanding of testing for the different parts of the SUT. Instead of considering how much of the SUT is covered by the test suite in total, it is considered how the different parts of the SUT are covered at the different levels of testing. A measure for the level of testing for the different parts of the SUT is defined and it is shown how this measure can be used during software maintenance and evolution to get a deeper understanding of the testing for the SUT and to evolve the existing test suite. The measure is applied to an open-source software (OSS) project to demonstrate its application in practice.

This paper is structured as follows. The next section discusses the basic concepts and describes the measure used in this paper. Section 3 shows how the measure has been implemented in practice. Section 4 applies the measure to an OSS project and analyses the results. Section 5 discusses the benefits and problems in applying the measure. Section 6 reviews related work in literature. Finally, Section 7 discusses conclusions and future work.

2. LEVELS OF TESTING

This paper uses the term *test granularity* to refer to the number of units of production code included in a test case (such as 'three methods'). The term *level of testing* is used to refer to a number of test granularity measures grouped together. For example, if we use the size 10 for a single level of testing, all tests with granularity 1–10 will belong to level 1 and all tests with granularity 11–20 will belong to level 2. If we use size '1' for the size of testing level, all tests will be mapped to the same testing level as their granularity (granularity 1 equals test level 1, granularity 2 equals level 2 and so on). However, also in this case, several tests can still be mapped to the same level if they have the same granularity. It is possible to vary this measure according to the goal of the analysis. The different concepts to be taken into account in defining this measure will be discussed in this paper.

2.1. Roles for the levels of testing

The role of testing in general can be defined as exercising the SUT with different inputs in order to reveal possible errors [1–3]. Test suites are composed of different types of tests, all of which have their own roles in testing and debugging the system. Rothermel *et al.* [4] have provided a survey of literature on advice about test granularity. This survey shows some contradictory advice on when to apply tests at different granularities. For example, Beizer [2] suggests that it is better to use several simple tests and Kit [5] suggests that large test cases are preferable when testing

Copyright © 2007 John Wiley & Sons, Ltd.

valid inputs for which failures should be infrequent. However, even though the advice on applying testing at different granularities shows some contradiction, there is generally an agreement on the roles of the different levels of testing.

Focused tests at a finer granularity run fast, focus on the cause of failure and make it possible to cover difficult paths of execution [1–4,6–8]. Thus, their role is best at verifying the finer details of a component's inner working and in debugging of faults. However, getting high coverage with small tests is expensive and verifying that individual components work in isolation does not tell whether they work correctly together. This means trade-offs need to be made in implementing focused tests.

Higher-level tests are required to verify the behaviour of the smaller parts as a whole and to validate the higher-level functions and properties expected from the system [1–4]. With higher level of testing it is less expensive to get a high test coverage as the tests cover larger parts. The trade-off is in verifying the finer details of the components and in debugging the cause of the failures. Covering complex details of small parts is difficult with large test cases and debugging can be time consuming when we only know that the fault is somewhere in the large portion of code executed.

For best results we need tests at lower and higher levels, where they are most useful. To be able to evolve the test suite and determine how the different parts of the code are exercised by test cases at different levels, we must be able to measure the levels of testing performed on the different parts of the SUT. As, during software maintenance and evolution, these parts and the test coverage of the regression test suite are likely to change, this analysis must be possible to be automated and repeated as much as possible. Measuring the test coverage at the different levels of testing is where the traditional coverage measures fail, as they do not give any information on how a piece of code is covered, only that it is covered in some way by the test suite.

To address this problem, this paper describes a technique for measuring how the different parts of the software are tested at different levels and builds on this to help make more informed decisions about where and how to focus future testing effort. However, before measuring these values, the measure of testing level and how it is related to the previous definitions in the literature is defined. Since the interest in this paper is in automating the measurement as far as possible, the definitions are reviewed from the viewpoint of how they can be measured automatically from test execution.

2.2. Defining the levels of testing

In the traditional testing literature, testing is divided into two basic types of testing: white box testing and black box testing [1,2,5]. These are further divided into more specific types, so that white box testing typically includes unit testing and parts of integration testing and black box testing typically includes acceptance tests, functional tests, system tests and higher-level integration tests. White box tests are typically considered to be lower-level tests and black box tests to be higher-level tests.

For quantitative measurement of test granularity, these types of classifications are problematic. The scope of a unit in a unit test can be defined to be, for example, a method, a class, a cluster of classes, a subroutine or a subprogram [1–3,5,9]. Similarly, integration testing can combine any number of these different units. By these definitions both unit tests and integration tests can include different sizes of groups of methods or classes in the SUT. Thus, by looking at the code executed by a test it is not possible to tell when a test stops being a unit test and becomes an integration test, or the other way around. Similarly, black box tests can exercise a small or large amount of code depending on how the tested functionality is spread in the code.

Copyright © 2007 John Wiley & Sons, Ltd.

Rothermel *et al.* [4] have used a definition of granularity based on the test case input. Their measure of granularity is based on the size of the test cases, with the size being measured by the number or amount of input applied per test case. A test case with more input is a test case of a higher granularity than a test case with less input. This measure can be used to classify the test cases by their granularity. However, this measure does not tell us anything about the size of code executed by a test case. The amount or number of input is not tied to the size of code used to process it, as small or large amounts of input can be processed by small or large amounts of code.

This paper defines the testing level by giving each test case a numerical measure based on how many different units of code, such as classes, methods or lines of code (LOC), are exercised in the test case. Similarly, any measure central to a system's functionality, such as messages or events in a message- or event-based system, could be used. Any of these can be used to define the test granularity and thus the level of testing for the code exercised by the test case. This gives a quantitative, automatically measurable, measure of the granularity of each test case, which can then be used to evaluate the levels of testing for the different parts of the SUT. For example, when using the detail level of methods, when a test case exercises code from 10 methods its granularity is 10. When it exercises code in 20 methods, its granularity is 20. Once these granularities are mapped to testing levels, these levels can be ordered and compared for all parts and systems as long as the same measure of classes, methods, LOC or combination of these is used for each test.

3. MEASURING THE TESTING LEVELS

The measurement data for the testing levels described in this paper are gathered in two steps. In the first step, all the test cases for the SUT are executed and the coverage information for each test case is gathered. The coverage information provides the granularity of the test cases, which is needed for the second step. In the second step, the level of testing for the different parts of the system is calculated. This process is described in more detail in this section, starting with step 1 and followed by step 2.

3.1. Measuring the granularity of test cases

The components and the process used to gather the data for the first step are described in Figure 1. It describes an implementation for the Java platform as used in this paper. The used JUnit [10] and AspectJ [11] components are freely available OSS components and the measurement can also be implemented on any platform that has similar components available. Other approaches to collect the execution traces of the test cases also include tracing through special debugging interfaces provided by the platform [12] or using a common code coverage tool to measure the coverage for each test case [13]. As a data store it is possible to use, for example, the file system or a database. Both were successfully prototyped for this paper.

The production code is first instrumented to produce trace events for all method calls. Then, the test cases are iterated and coverage data for each test case are collected, until all test cases have been executed. This measurement can provide data for telling which LOC were executed, which methods were called and which classes were used in each test case. Here AspectJ has been used for tracing, which provides support for custom trace code. The used detail level of method calls

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

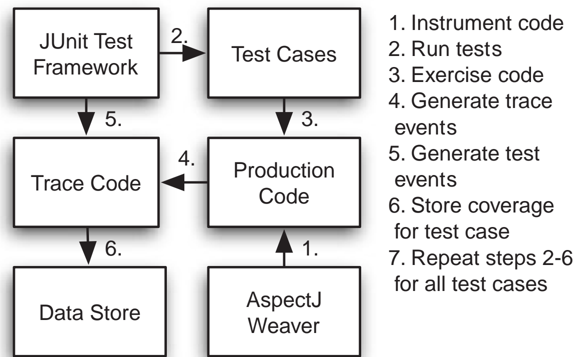


Figure 1. Measuring data for step 1.

provides a compromise between very fine detail (LOC) and coarse detail (classes). While using aspects to trace at the LOC level is not straightforward, it is possible to implement LOC level measurement by using a code coverage tool and this was successfully experimented with during this study. For different interests it is possible to vary the level of detail used while the rest of the process remains unchanged. Once these data are collected, it is possible to move on to the second step.

3.2. Associating the tests with tested parts

The second step is illustrated in Figure 2. This figure shows a simple example system consisting of four methods in two classes and four test cases. By having measured which parts of the code are executed by which test cases, we have collected the information presented in the figure. Test case granularity is calculated by adding up the number of methods executed by each test. This information is shown in Table I. The number of methods tested at the given granularities are calculated simply by adding up the unique methods covered by tests at given granularities. This information is shown in Table II.

While the figure shows the associations between the tests and the methods, the actual path of execution can be anything as long as the method is executed as a part of running the test. Where the methods are invoked from makes no difference, as the measurement system will record any call to the observed methods while the test case is executed. It can be invoked, for example, from the test case or from any other production code. The set of observed methods can be limited, for example, by instrumenting only the parts of interest for coverage or by filtering the collected data.

Once we have associated each test case to the code it executes, we can calculate the metrics on how each method is tested. For example, to calculate the lowest level of testing for each method, we first find the smallest granularity from the test cases associated with the method. This tells us the most focused test case for that part of the code. Once this is known, the granularity value needs to be mapped to the testing levels, which shows the lowest level of testing for that method. Similarly, it is possible to get the highest level of testing by finding the maximum associated value.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

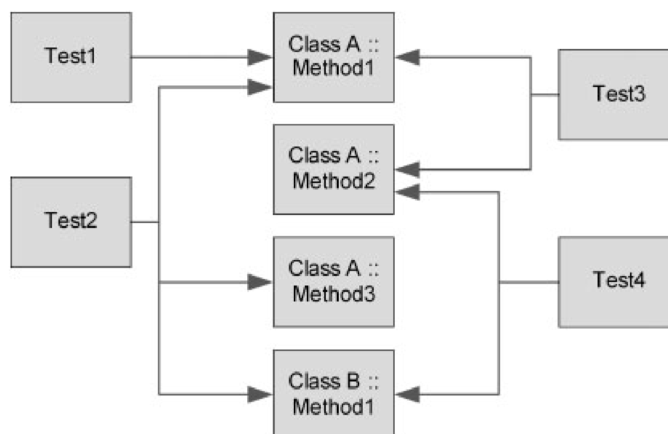


Figure 2. Associating tests with the parts of software executed.

Table I. Test granularities.

Test	Granularity
Test1	1
Test2	3
Test3	2
Test4	2

Table II. Method count at different granularities.

Granularity	Number of methods
1	1
2	3
3	3

Table III. Test granularities for the methods.

Method name	Min	Max
Class A::Method1	1	3
Class A::Method2	2	2
Class A::Method3	3	3
Class B::Method1	2	3

For Figure 2, using testing level size 1 (mapping the test granularity directly to the same level), the lowest and highest testing levels are shown in Table III. For example, `Class A::Method1` is associated with test cases Test1, Test2 and Test3. As Test1 has the smallest granularity of these,

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

the lowest level of testing (min) for this method is 1. As Test2 has the highest granularity (3) of these, the highest level of testing (max) is 3. Similarly, `Class B::Method1` is associated with test cases Test2 and Test4. Thus, the min and max values are accordingly 2 (Test4) for the lowest level and 3 (Test2) for the highest level.

4. EVALUATING AND EVOLVING A TEST SUITE

As an example of applying the technique, the test suite of PMD [14], an OSS Java source code analysis tool, is analysed in this section. To help put the measurement data in context, the total number of tests, classes, methods and source LOC (SLOC, LOC excluding whitespace and comments) for the project are shown in Table IV. To collect the coverage information for the testing levels, the test suite for the project has been executed, the granularity of all test cases has been measured and these data have been mapped to the methods in the SUT as described in Section 3. Thus, the information needed to calculate the different levels of testing performed for all methods in the SUT is available.

4.1. Testing levels—an overview

To get an overview of the testing at the different levels, the first step is to look at how much of the SUT has been covered at the different levels. This provides a basic overview of the testing done at different levels as a total and shows whether there is, for example, a lack of low- or high-level testing in general. Using the overview data as history information also makes it possible to track the evolution of the levels of testing over time. However, as discussed earlier, taking the analysis of this testing data further poses the question of what size to use for the testing level. If we simply use size 1 for each testing level (mapping the granularities directly to test levels), the overview will describe too many details and not give the high-level overview we are interested in. As an example, Table V lists the number of PMD methods tested at granularities 1–10 and 21–30.

At the lower granularity of 1–10, we see a large number of tests and methods covered at each granularity. However, as we move to higher test granularities, we start to see a higher spread of the tests as shown already by the tests at granularity 21–30. Here there are at most two tests at a given granularity. This spread of tests is further amplified the more we move towards the higher granularities. The highest granularity for a single PMD test case is 834. The complete spread of the number of tests at different granularities is illustrated in Figure 3, which shows a histogram

Table IV. Project metrics.

Metric	Total
Tests	781
Classes	629
Methods	4073
SLOC	38 790

Copyright © 2007 John Wiley & Sons, Ltd.

Table V. PMD Number of methods (NOMs) covered and number of tests (NOTs) at levels 1–10 and 21–30.

Level	Number of methods	Number of tests
1	9	14
2	44	39
3	39	17
4	50	16
5	50	22
6	38	8
7	58	17
8	24	3
9	64	13
10	95	12
21	23	2
22	44	2
23	23	1
24	25	2
25	25	1
26	0	0
27	27	1
28	0	0
29	58	2
30	30	1

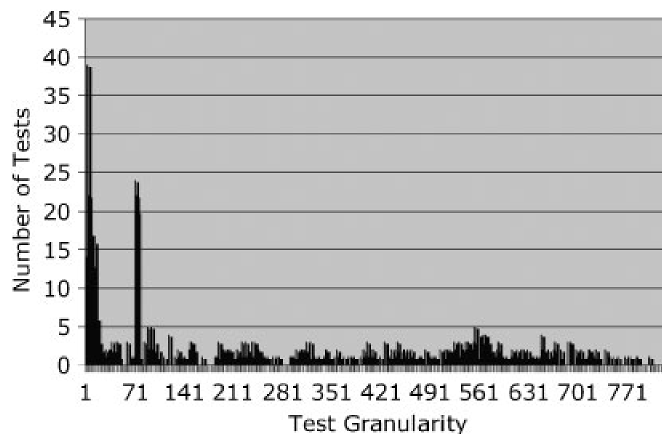


Figure 3. Number of tests at different granularities.

of the number of tests at different granularities. Figure 4 shows the number of methods covered at each granularity. Figure 5 shows how the data can be summarized to describe tests at multiple granularities to single testing levels, and to provide a better high-level overview.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

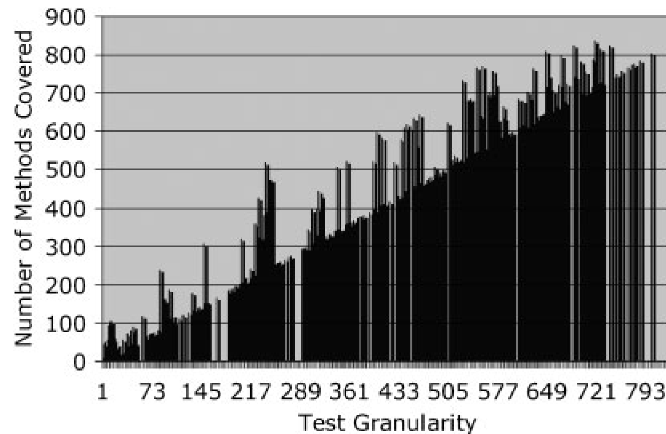


Figure 4. Number of methods covered at different granularities.

In Figure 5, the data for different testing levels using different level sizes are presented. In the first row, the level size is 10, in the second 50 and so on. For example, each row for levels 1–10 in Table V is represented by a single bin (for a total of 10 bins) in Figures 3 and 4. However, in the figures in the first row in Figure 5, a single bin represents this information for levels 1–10. The level range then contains the data for 10 test granularities as one level. The different level sizes provide higher abstraction overviews of the coverage at the different testing levels. For example, in Table VI we see that, when using a level size of 1, it seems that 471 methods are covered by tests at a granularity of 1–10. But when combined and viewed with a level size of 10, we see that this only includes 274 unique methods. This is due to the partial overlap of the different methods being covered in multiple tests at adjacent granularity.

When considering the roles of the test cases for the different levels, it does not make much difference whether a method is covered by a test case of granularity 1 or 10. For covering critical parts and making debugging easier, we may be interested in ensuring that we have good coverage at finer levels, but debugging 10 methods should still be relatively easy. Our interest for the size of the viewed levels can vary according to what we are looking for. At lower levels a finer spread with a ratio of 10 or 50 may be appropriate. On the other hand, at higher levels, we may only be interested in some form of higher-level coverage and may use a ratio of, for example, 200 or may even look for any coverage with tests over a certain threshold granularity, such as 100.

While in these examples we have viewed the whole project at once, all these analyses can also be applied to smaller parts of the system. If we consider, for example, parts of the system to be more critical or error prone, we can filter only these parts of the code for analysis. This can be based on concepts such as project structure, domain knowledge or different complexity metrics. However, this is considered out of the scope of this paper and left as a topic of future work.

The analysis presented so far is focused on an overall view of the software testing in general. However, to more concretely and further evolve the test suite and the tested code, more detailed analysis is needed. This will be looked into next.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Range	Number of tests/Testing Level	Number of methods covered/testing level
10		
50		
100		
200		

Figure 5. Spread using different test level sizes.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VI. Method counts for different level sizes.

Level size	Range	Number of methods
1	1–10	471
1	1–50	1896
10	1–10	274
10	1–50	1040
50	1–50	603

4.2. More detailed analysis—evolving the tests and code

Analysing in detail the way the production code is covered provides opportunities for both optimizing the test suite and optimizing the production code. We can improve the test suite by implementing new test cases at different levels where needed, and by removing and combining overlapping tests. We can improve the production code by identifying the redundant code and, by adding new tests to identify possible problems, increasing our confidence on code quality. As we are interested in more detailed analysis, the view in this case is also focused on more detailed analysis. Thus, in this subsection, the test granularity is also used as the testing level, which gives the most detailed information available.

From the overview analysis it is possible to find interesting focus areas for analysis. For example, one interesting aspect from the overview at the finest level presented in Figure 3 is the peak at levels around 70. The detailed information for this data is presented in Table VII. The interesting aspect of this data is that, for each granularity in 70–73, there are exactly the same number of unique methods covered as the test granularity. As there are a number of tests at each of these granularities, it means that all these tests at the same granularity are executing the same methods. Thus, the tests are simply exercising the exact same functionality with different inputs. With up to 24 tests at a single granularity all exercising the same functionality, these tests provide good candidates to consider for trimming the test suite for excess tests by removing or combining separate test cases.

To find the methods that should be considered for further testing at a lower level, the methods in the chosen part of the SUT must first be analysed to see which of them are not tested at a lower level. To find which methods are only exercised as part of high-level tests and not tested at a fine granularity, for each method, the lowest level of testing is first taken for analysis. By ordering all the methods by this level (a simple sorting), all that needs to be done is to look at the methods with the highest values and these are the methods to be considered first. A sample of these top candidates is shown in Table VIII.

To conserve space, the table lists only one method from a class if there are multiple methods in the same class at the same testing level. For example, there are three methods (CM in the table) in the class `ast.JavaParser` that are each tested at the finest level (Min in the table) as a part of a test case which exercises 834 methods. As these metrics describe some of the largest test cases in the test suite, and variation in test size at this level is high, the methods in the same class and at the same level are likely executed in the same test case. As a note on the different methods, by looking at the source code and its comments, it is clear that all the code in the `ast` package is generated by a parser generator. Thus, the `ast` package code is not considered here for further testing as it consists solely of generated code, and testing it would mean testing the parser generator.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VII. Data for levels 69–74.

Testing level	Number of tests	Number of methods
69	0	0
70	24	70
71	6	71
72	22	72
73	20	73
74	0	0

Table VIII. Largest minimum levels for methods.

Method name	Min	CM
ast.JavaParser.jj_3R_120()	834	3
strings.ConsecutiveLiteralAppends.getSwitchParent(Node, Node)	802	9
rules.ConstructorCallsOverridableMethod.MethodInvocation.isSuper()	785	37
design.ImmutableField.inLoopOrTry(SimpleNode)	773	6
strings.InsufficientStringBufferDeclaration.isLiteral(String)	768	11

To consider these methods further, the roles of testing at different levels need to be considered. The roles defined for lower-level testing in Section 2 were aiding in debugging and verifying the finer details of the method's inner workings. Thus, for debugging, if there is a fault in any of these methods, it will be much more difficult to find the cause of failure as the failure will only show as a part of a large test. For example, if there is a fault in the method `ConsecutiveLiteralAppends.getSwitchParent(Node,Node)`, the best indicator is a test that exercises 802 methods. Thus, finding the cause of failure requires looking into all these methods. Also, if a method tested only at this level provides complex behaviour, the finer details of this behaviour are unlikely to have been tested well. By looking more closely at these methods, their intended behaviour, usage and similar properties, it is then possible to assess whether the methods should be considered for inclusion in new test cases.

When looking for methods that need testing at a higher level, the methods in the chosen part of the SUT can be ordered by their highest testing levels. From these, we look for the methods with the smallest values to find the ones to consider first for implementing new higher-level tests. A sample of the top methods tested only at a low level is shown in Table IX. In Section 2, the roles for higher level of testing are listed as verifying the working of the smaller parts as a whole and as verifying the higher-level functionalities of the program.

As classes and methods in a program should be implemented to be a part of a larger piece of functionality, there should be test cases that also make use of each class and its methods in a larger context. Thus when parts are only tested at a low level, this could highlight missing testing for a higher functionality of the SUT or even possibly a class that has become redundant and is no longer needed or used elsewhere in the system.

The final decision of removing a code that is considered redundant should be left to a maintainer with expert knowledge of a system. However, here a feature common in today's integrated

Copyright © 2007 John Wiley & Sons, Ltd.

Table IX. Smallest maximum levels for methods.

Method name	Max	CM	*U*
symboltable.TypeSet.getASTCompilationUnitPackage()	2	2	1
stat.Metric.getTotal()	2	2	1
ant.Formatter.setToFile(File)	2	1	1
strings.AvoidDuplicateLiteralsRule.ExceptionParser(char)	2	2	6
pmd.CommandLineOptions.usage()	2	1	3

Table X. Most tested methods.

Method name	Number of methods	Range
ast.SimpleNode(int)	569	5–834
symboltable.SourceFileScope(String)	429	13–834
report.PackageNode(String)	374	2–834
pmd.RuleContext.setSourceCodeFilename(String)	350	3–834
symboltable.ScopeAndDeclarationFinder.cont(SimpleJavaNode)	331	118–834

development environments such as Eclipse [15] and IntelliJ [16] has been applied: finding the usages of a method or a class in the source code. The *U* column in Table IX lists the results for finding the usages for the listed methods. While in this paper this analysis was applied only to the few methods listed manually, it could easily be automated with existing analysis tools. For example, the analysis showed that the method `Metric.getTotal()` is used only in a single test case that does nothing but test this single method's functionality. The method is not used in any of the production code, but looking at the traditional code coverage view would show it as covered, while in fact it is not used in any production code.

One more interesting aspect to look at as a side effect of this analysis is the summary of how many tests are exercising different methods. This information can help both in understanding the system implementation and in finding the most critical parts of the system for testing, both important concepts in software maintenance and evolution. The more the tests exercise a method, the more central that method is to the system's functionality. Table X lists a sample of the top most-tested methods in the system. For example, the method `ast.SimpleNode(int)` is executed in 569 different test cases. These test cases range in size from a granularity of 5 to a granularity of 834.

Here only one method has been picked from the `ast` package, but overall, out of the 500 most tested methods (ranging from methods being executed in 200–569 test cases), 372 belong to the generated `ast` parser package. Since `ast` is a structure used to describe source code and PMD is a source code analyser, it is quite clear that this is and should be a central concept in the system.

An example of a method that is central to the functionality but is not tested at a finer level is `ScopeAndDeclarationfinder.cont(SimpleJavaNode)` shown in Table X. This method is executed in 331 tests, but is at the finest granularity in a test that exercises 118 methods. Thus, this metric could also be used to aid in locating new test subjects, in addition to metrics such as code complexity as proposed in the previous section.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

5. DISCUSSION

In this paper, the presented measurement technique has been applied to an OSS project for which no detailed information was available. When using the technique for a project we are developing, we know the SUT better and applying the technique and analysing the results is easier. However, the success of applying the technique on a project without detailed knowledge of the SUT shows the technique to be applicable in practice. It was possible to get an overview of how much of the SUT is covered at the different levels and highlight places in the SUT to consider for further evolution of the production code and the test suite. As the technique does not consider untested parts of the code, these have to be first brought under test to be included in this analysis. Traditional code coverage measures and other existing techniques can be used for this purpose.

Once the detail level of interest for the overview analysis is found, observing the evolution of the different levels of testing over the history of the project can be used to monitor the testing process. If we set a goal to get more of the SUT covered at a higher or lower level, we can then use the overview to observe how this goal is met by looking at the evolution over time. This can be useful for management purposes and to monitor our own progress as we work towards the goal of coverage at different levels. However, it should be kept in mind when considering this overview that, although the levels of testing tell more about the testing over different parts of the SUT, it still does not tell whether the tests at the different levels would be comprehensive and good. It makes one aspect of test case properties visible, but does not mean that full coverage at different levels would mean perfect testing.

Measuring the granularities of testing and mapping these values to the different parts of the SUT to get their levels of testing can be automated as is done in this paper. Different aspects of analysing the results can also be automated as was demonstrated by using existing tools to find method usages. However, detailed and final analysis of these results still needs human work. Tool support to aid in this can be further developed by using, for example, complexity measures or measures to find aspects of method importance, for which one metric was shown in this paper.

Using the technique to find places lacking in different levels of testing can have several benefits, as shown by the analysis of the OSS project in Section 4. Finding where there is a lack of higher-level testing can bring out untested higher-level functionality. All code in a software system should exist to help implement the higher-level functionalities required by the system and as such take part in higher-level tests. However, not all code needs to be tested at higher levels as some code can be required by, for example, programming language constructs for exception handling or similar reasons and be untested as part of higher-level tests. Similarly, as the measure of level of testing in this paper is based on the size of code executed, higher-level functionality can be implemented as part of small or large amounts of code. As such not all parts that are only covered by what is measured to be a low-level test necessarily need to be made part of a higher-level test. This highlights a topic that needs more research and shows how the technique is best used as a tool to help in analysing the test suite by a human analyst who can judge where new tests are actually needed.

In a larger context, as the technique described in this paper provides a quantitative measure of the levels of testing for the different parts of the SUT, it enables doing more research on the levels of testing. Using the technique, it is now possible to see how the different parts of the SUT are covered at different levels and use this information to analyse, for example, different implementations of testing levels and their correlations with other properties of the tested parts of the SUT. This is

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

where the most detailed information provided by the technique can be most useful as it allows doing the most detailed analysis of these properties. This can be especially helpful in instances of software evolution.

6. RELATED WORK

Regression test selection and optimization are research topics that focus on choosing which tests from a test suite to run [17] and optimizing their order of execution [18]. In these cases, the granularity of test cases is considered with such goals as how to maximize the coverage fast or how to get additional coverage. The effects of test suite granularity on the cost-effectiveness of regression test selection, optimization and reduction have been studied by Rothermel *et al.*, who focus on the granularity as defined by the test case inputs [4].

When these studies on regression testing consider test granularity, they measure it either as code executed or by the size of input in each test case. The executed code is not used for measuring the granularity of testing, but rather for finding a minimal set of tests to provide maximum coverage. This paper measures the code executed by each test case and, in addition, applies a second step of measurement, where the test granularities are mapped to the code to measure the level of testing for the different parts of the tested code. Another difference is in the optimization goal; whereas these studies focus on optimizing the execution of existing tests, this paper focuses on optimizing the implementation of further test cases.

Zeller and Hildebrandt [8] and Chesley *et al.* [19] have developed methods and tools for finding the cause of failure from coarse-grained tests which execute large parts of the SUT. A failing test case is executed repeatedly with varying input or code changes until the smallest part that causes the failure is found. Both of these techniques can lessen the need to implement lower-level testing; however, as also noted by Gälli *et al.* [7], having finer granularity tests can make these techniques work faster. Also, we still need different levels of testing to verify the finer-level details and the higher-level functions. In many cases it is also much faster to debug something if we have focused tests where we want instead of having to run specific tools and methods to filter out the cause.

Nagappan has developed his own metrics suite, called Software Testing and Reliability Early Warning (STREW) metrics suite, for predicting software maintenance and guiding the testing efforts [20]. STREW is based on a number of metrics measured from both test code and production code, such as number of assertions, complexity and coupling. The approach applied in STREW is similar to that in this paper, in applying measurement to the testing and production code to guide the testing process, but his metrics suite does not consider test granularity.

Pighin and Marzona [21] propose to focus the highest testing effort on the most fault-prone parts of the software. They argue that it is a waste effort to put the same effort of testing on the less fault-prone parts of the software as on the more fault-prone parts. Their approach is common with this paper in that it proposes a method to focus the testing effort and uses the properties of code to guide this process. However, they discuss allocating time, not how to focus the testing on the fault-prone parts. The information in this paper provides a way to help focus the extra effort spent on the chosen parts.

Copyright © 2007 John Wiley & Sons, Ltd.

Jones *et al.* have developed a technique and a tool for visualization of test information to assist in fault localization [22]. Their technique colour codes source lines based on their execution in passed or failed test cases. This is based on gathering coverage information for every executed test and mapping the executed passed and failed tests for each LOC. This way, colour spectra can be applied for each LOC to give it a colour based on how many failed tests are executed in that line. The authors then propose that specialists can use this information to help debug the faults. This approach uses a similar mapping of tests to code as is done in this paper, but, while they use it for counting the number of failed tests for each line, they do not consider the granularity of testing or the roles of the testing. Instead, their focus is on highlighting where the possible failed statement is and using it for debugging.

Baudry *et al.* [12] define a test criterion for improving debugging, called test-for-diagnosis (TfD) criterion. A good TfD value is defined as maximizing the number of dynamic basic blocks (DBBs). DBB is defined to be the set of statements covered identically in test cases. Using their own test suite optimization algorithm, Baudry *et al*. optimize existing test suites for TfD. Their aim is to optimize the test suite to make debugging faults easier. To assist in this, they use the localization technique proposed by Jones *et al.* [22], in which the test suite is optimized using the TfD measure. Their DBB measure can be considered as a form of granularity measure, but is not usable for the purposes of this paper, as the size of DBB varies and thus any granularity measured with it would not be comparable. In line with this, they do not provide means to assess the granularity of testing for the different parts of the code, but focus on the debugging of failed test cases.

Sneed [23] has used both static and dynamic analysis for linking test cases and use cases to the code they execute. He started with static analysis, finding it inadequate for his purposes, and then moved to dynamic analysis, similar to this paper. While he used timing-based matching to match test cases, this paper makes use of instrumenting both the test framework and the code under test to automatically link the test case execution to the code under test. Similarly, he used static analysis techniques to map the trace data to the functions executed, while in this paper the information is provided directly by the trace framework (AspectJ). This is mostly a function of different environments and both types of tracing have advantages in different environments. Finally, while he focused on using the information for regression test selection, this paper has focused on understanding the test suite and its composition. However, the data could also be applied to regression test selection in a similar way as Sneed has done.

Advances in coverage-based tools are moving them to also include more detailed coverage information at the individual test level and using different coverage measures such as method, block and predicate coverage [13]. While these tools do not yet provide a deeper analysis of the coverage information as presented in this paper, extending them with this support should be simple as they already provide the basic individual test case coverage information needed to perform the analysis.

7. CONCLUSIONS AND FUTURE WORK

This paper proposed a technique for measuring and optimizing the levels of testing over the different parts of the system under test (SUT). It was shown how this can be applied to support software maintenance and evolution by showing how to measure the levels of testing for the different parts of the SUT, how to get an overview of the total testing over the SUT and its smaller parts and how

Copyright © 2007 John Wiley & Sons, Ltd.

to use this information to further evolve the production code and the test suite. Different levels of testing have different roles in testing a system; having tests at these different levels makes it more likely to find faults earlier and makes debugging them faster. Full coverage at these different levels would be optimal, but it is always a trade-off and choices have to be made. This technique helps make these choices more explicit by showing how the different parts of the SUT are tested at the different testing levels. The technique was applied to an OSS project to illustrate its use in practice.

In summary, the technique presented helps in finding the following:

- Untested higher-level functionality by highlighting places in the SUT lacking in higher-level testing. All code should serve to implement the required higher-level functionality of the system and thus take part in higher-level tests.
- Redundant code that is no longer needed, by finding parts that are not tested at a higher level and are no longer needed for any higher-level functionality.
- Parts of the SUT that are lacking in different levels of testing, for example, parts in need of low-level testing to help in debugging or for verifying complex behaviour.

In addition, the technique provides possibilities to

- get an overview of the testing done at different levels over the SUT;
- find and understand the central components in SUT implementation;
- track evolution of the test suite and the SUT with regard to test levels; and
- do research on different levels of testing by providing an automated, quantitative measure.

Further research to improve the use of the technique would include developing techniques to help filter out the information of interest, including the most important parts to consider for further testing, and to study the optimal distributions for the levels of testing for different methods. Studying the properties of the source code with relation to the different levels of testing is also needed to bring out the possible trade-offs in implementing tests at different levels. For example, it is not always possible to test every part at a finer granularity if they are highly coupled or getting lower coupling may bring higher complexity. While all parts of the techniques implementation and data analysis can be automated, integrated tool support is also still needed for enabling practical adoption.

REFERENCES

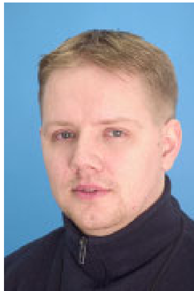
1. Binder R. *Testing Object Oriented Systems*. Addison-Wesley: Reading MA, 2000; 1200.
2. Beizer B. *Black Box Testing*: *Techniques for Functional Testing of Software and Systems*. Wiley: New York NY, 1995; 320.
3. Myers G, Badgett T, Thomass T, Sandler C. *The Art of Software Testing*. Wiley: New Jersey, 2004; 256.
4. Rothermel G, Elbaum S, Malishevsky AG, Kallakuri P, Quit X. On test suite composition and cost-effective regression testing. *ACM TOSEM* 2004; **13**(3):277–331.
5. Kit E. *Software Testing in the Real World*. Addison-Wesley: Reading MA, 1995; 272.
6. Feathers M. *Working Effectively with Legacy Code*. Prentice-Hall: Upper Saddle River NJ, 2004; 456.
7. Gälli M, Lanza M, Nierstrasz O, Wuyts R. Ordering broken unit tests for focused debugging. *Proceedings 20th IEEE International Conference on Software Maintenance* (*ICSM'04*), 2004; 114–123.
8. Zeller A, Hildebrandt R. Simplifying, isolating failure-inducing input. *IEEE Transactions on Software Engineering* 2002; **28**(2):183–200.
9. Runeson P. A survey of unit testing practices. *IEEE Software* 2006; **23**(4):22–29.
10. JUnit: Testing framework. http://www.junit.org [3 July 2006].
11. AspectJ: Java programming language aspect oriented programming extension. http://www.aspectj.org [6 May 2007].

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

12. Baudry B, Fleurey F, Traon YL. Improving test suites for efficient fault localization. *Proceedings of the 28th International Conference on Software Engineering* (*ICSE'06*), 2006; 82–91.
13. Lingampally R, Gupta A, Jalote P. Multipurpose code coverage tool for java. *Proceedings of the 40th Annual Hawaii International Conference on System Sciences* (*HICSS'07*), 2007; 261.
14. PMD: Source code analyser. http://pmd.sourceforge.net [6 May 2007].
15. Eclipse: Integrated development environment. http://www.eclipse.org [6 May 2007].
16. IntelliJ: Integrated development environment. http://www.intellij.com [6 May 2007].
17. Rothermel G, Harrold MJ. Analysing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; **22**(8):529–551.
18. Rothermel G, Untch RH, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948.
19. Chesley OC, Ren X, Ryder BG. Crisp: A debugging tool for java programs. *Proceedings 21st IEEE International Conference on Software Maintenance* (*ICSM'05*), 2005; 401–410.
20. Nagappan N. A software testing and reliability early warning (STREW) metric suite. *PhD Dissertation*. North Carolina State University, 2005; 136.
21. Pighin M, Marzona A. Optimizing test to reduce maintenance. *Proceedings 21st IEEE International Conference on Software Maintenance* (*ICSM'05*), 2005; 465–472.
22. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. *Proceedings of the 4th International Conference on Software Engineering* (*ICSE'02*), 2002; 467–477.
23. Sneed H. Reverse engineering of test cases for selective regression testing. *Proceedings of the 8th European Conference on Software Maintenance and Reengineering* (*CSMR'04*), 2004; 69–74.

AUTHOR'S BIOGRAPHY



Teemu Kanstrén is a Research Scientist at VTT Technical Research Centre of Finland. Prior to joining VTT he worked as a Systems Analyst in the software industry for 5 years. He is currently a PhD student at the University of Oulu, where he also received his MSc in Computer Science in 2004. His research interests include automated software testing and analysis.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

PAPER IV

A Study on Design for Testability in Component-Based Embedded Software

In: Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications, SERA'08, Prague, Czech Republic, 20–22 August, 2008. 8 p.
© 2008 IEEE.
Reprinted with permission from the publisher.

A Study on Design for Testability in Component-Based Embedded Software

Teemu Kanstrén
VTT, Kaitoväylä 1, 90571 Oulu, Finland
teemu.kanstren@vtt.fi

Abstract

Effective implementation of test automation requires taking testing into account in the system design. In short, this is called design for testability (DFT). In this paper a study on DFT in component-based embedded software is presented, based on the interviews and technical documentation from two large-scale companies in the European telecom industry. The way test automation is addressed and the different techniques applied to make this more effective at the architectural level are described. The differences and benefits of different approaches are discussed.

1. Introduction

Effective implementation of test automation requires taking testing into account in the system design. In short, this is called design for testability (DFT). This paper presents a study on DFT in two large-scale companies in the European telecommunication industry, both working on similar products, based on the same standards. The way software (SW) test automation is addressed and the different techniques applied to make this more effective at the architectural level are described. The differences and benefits of different approaches are discussed.

The testing discussed is different levels of black-box integration testing. At the lowest level, small components are composed together to larger components, and the internal messages between these components are considered. Properties such as internal structure at the level of code are not considered. At the highest level, all components are fully integrated as a complete system. In each case there is a separate test team dedicated to testing the components/system. For debugging problems, analysis is done at a more detailed level including the use of white-box techniques.

The tested systems are large-scale telecommunication systems. Each system consists of a number of hardware (HW) blades running SW with different functionality. Additionally, each system also interfaces with a set of other standardized systems, and both the internal correctness needs to be tested as well as the external interactions. Different parts of the SW are implemented in different programming languages, such as C/C++ and SDL. The system is divided to different sizes of components, and the testing described is done at the level of these components. While the system size in terms of lines of code was not always given, for example one system consists of about one million lines of production code, with a similar amount of code for the test environment.

As these are embedded systems, external HW measurement devices can also be used. The instrumentation mechanism choice is a trade-off in minimizing the effects of monitoring (HW) and providing more sophisticated views into the systems (SW). In this paper the focus is on the SW solutions.

2. Design for Testability

The term testability in SW testing can be considered from various viewpoints [1][2][5][7]. While some consider the architectural viewpoints [8][9][10], few describe techniques for more effective DFT at the architectural level [1][3]. However, this is commonly identified as an important goal in SW testing research [4].

In this paper two main viewpoints of DFT are considered from the architectural viewpoint: controllability and observability [5]. To test a component, we must be able to control its input, behavior and internal state. To see how this input has been processed, we must be able to observe the components output, behavior and internal states. Finally, the system control mechanisms and observed data must be combined to form meaningful test cases for a system.

The definition of how the interviewed see DFT has some variation, but the basic concepts are similar. These different viewpoints include the possibility to simulate different parts of the system for testing, to isolate the part that is being tested, to control system behavior with specialized test functionality and to access information on system behaviour. When test code in integrated to observe or control a part of the system,

978-0-7695-3302-5/08 $25.00 © 2008 IEEE
DOI 10.1109/SERA.2008.11

the location of the test code is called a test point. When testing is run in a desktop simulation environment outside the target HW, this is called host testing.

The presentation and discussion of the test automation and DFT concepts in this paper are described according to the following main concepts: test implementation, control of messaging, simulation strategies and implementation of functionality to support testing. How the different companies address each of these concepts is described and discussed.

3. Research Methodology

The interviewed companies were chosen based on their mutual interest. As they were starting closer collaboration, both had interests to combine strengths of the two companies. The interviews followed a semi-structured format, where questions were grouped into themes. The goal was to allow the interviewed to freely express what they felt were important concepts, while keeping the focus on the matter at hand. The following is a list of the main questions addressed:

Theme	Main Question(s)
Test Automation	• How do you implement test automation? • What solutions do you use to support implementation of test automation?
Observability	• How do you collect information from your system? • How do you address constraints such as real-time requirements?
Controlability	• How do you support controlling system states, behavior and partitioning? • How do you focus on problem analysis?

In both companies, a number of specialists in test automation were interviewed. Each company was asked to select a number of specialists with good knowledge on the interview topics. Some technical documentation was also received, describing the test automation systems. Once the information had been collected, results were checked with the interviewed people.

4. Test Implementation

The basic test implementation in both companies is based on verifying the correctness of message sequences and checking of message parameters. Although the systems have hard real-time requirements, they are considered only at the system testing level and not on the integration testing level. While some load and stress testing is performed during integration testing, it mostly done in system level testing, as in these cases the complete system in composed and problems in high level integration and interoperability can be seen. For testing timing related functionality in integration testing, specific test cases are used that manipulate the timers used in the system. For example, they can be set to expire immediately to test timer related fault handling. The amount of generated test data can also be a problem, as the test bus can become exhausted, causing failures when buffers become full. This requires special considerations on how and where test data is processed.

4.1 Integration Testing

Both companies use basic test scripts to verify the message sequences during integration testing. Verification of message sequences is based on the external interfaces of components, which is seen to help shield the test cases from minor changes in system implementation. As these messages are captured at the component level, several thousand lines of code can be executed between messages. Typically, the information on internal messages is also available, but these are only used when problems are found and need to be analyzed. Failing test cases are executed with more detailed logging to focus on the cause of failure, including internal messages passed and their parameter values. Most difficult problems to debug are seen to be problems that come up during long uptime, slowly consuming resources such as memory or CPU load.

Company 1 (C1) has used a traditional approach of developing test components (stubs) as needed in isolation. Company 2 (C2) has taken a different approach where, during development and integration testing, two versions of the system specification are implemented. One is the actual product and one is the test system. The test system provides simulated versions of all the components in the production system, and is developed using similar development and quality assurance processes as the production system. As two versions of the specification are implemented, they provide validation for each other and the understanding of requirements. In case of a failing test case it is necessary to consider which implementation is wrong, the test system or the production system. As same development processes are used for both systems, the same metrics can also be collected and compared for both. This can provide interesting insight into the effectiveness of test automation development. For good test system implementation it is seen that a ratio of 1 to 1 is good and a ratio of 1 production system error to 2-3 test system errors is more common.

4.2 System Testing

Similar to integration testing, C1 has relied on scripting of input and output also at system test level. C2 used to do the same but has moved to using higher-level abstractions due to difficulties in maintaining the test suites. In this case, the message sequences are encapsulated inside test building blocks, which describe high-level functionality of a system. These are further grouped into test cases, which are grouped into test suites. When system functionality is changed, updating test cases requires changing only some of the building blocks and not all test cases. As the blocks can be further reused over a product family, this has been found to lead to lower maintenance costs. Also, as test cases can be built from higher level abstractions (building blocks), it is easier for a system tester to write test cases without detailed knowledge of system internals. The goal is then similar to approaches such as model based testing [12], with the aim of using a higher level model abstraction to build test cases. In this regard, the implementation of the C2 system test environment takes more effort, but has smaller maintenance effort as changes are contained in the shared test components.

As described earlier, most of performance and load testing is left for the system testing phase. However, while C2 has put more effort into creating an advanced functional test environment for system testing, they have used only basic timing measurements of external interfaces also at the system testing level. For this type of testing, C1 has put more effort on advanced techniques for supporting analysis of resource usage and performance. This is based on analysis of detailed internal information, starting from generic properties and progressing to more detailed analysis based on the findings from the generic properties. At this level, the parameters include task switches, data on resource usages and use of OS services. As these can be monitored from outside application code (at the system level), they do not require as large effort to implement. When more focused information about an identified problem area is needed, more specific tracing is implemented. This data includes properties such as component inputs and outputs, system id values and data streams. Further, analysis tools have been developed to analyze this information using multivariate analysis techniques. This has been found very useful in system optimization.

5. Control of Messaging

Effective implementation of test automation requires being able to control the system execution and observe the results. In this regard, it must be possible to create different compositions of the system and its components, including the use of test components (stubs) as replacements for actual components. In component-based SW, a common means to compose components together into larger systems is through middleware [11]. Both C1 and C2 have taken a similar approach to make this possible, by controlling the messaging between the components, through their middleware. This section reviews these approaches.

5.1 Company 1

C1 uses a commercial off the shelf (COTS) third-party operating system (OS), targeted especially at embedded systems, in their products. For enabling creation of system test compositions and the use of simulation, rerouting of the system communication mechanism is used. In system execution, the execution of components is mapped to the OS processes, which run them as tasks. All communication between components is done through the system messaging interface, which is an inter-process communication (IPC) interface. The communication is handled by a system internal routing component that delivers the messages to the correct processes and components. As the same messaging interface is used over all the components and is based on a standard protocol, it is easier to build generic and reusable test services for this interface.

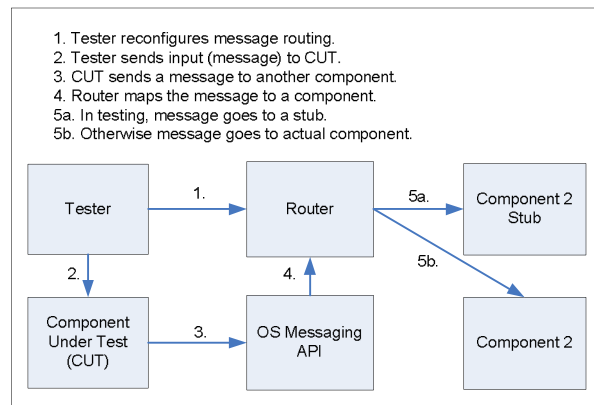


Figure 1. C1 message flow control and testing.

The process of using this mechanism to attach simulated components to the test target is shown in Figure 1. The system router component contains a routing table for passing the internal system messages to different components. By modifying this routing table, the messages can be passed to test components instead of production components. In addition to basic test stubs, which provide messaging functionality, this has also been used to implement more complex functionality to gain control over deeply embedded functionality. This is discussed in more detail in section 7.

5.2 Company 2

As a basis in their products, C2 uses a generic open source software (OSS) OS, which can be used equally well in both a host test environment and on embedded target HW. The enabler for using the test environment is the custom middleware on top of which the whole system runs. This middleware contains a communication translator component, which handles the addressing of component communication. When the system components are composed together to form the system, each SW component publishes over the communication translator their communication id values and subscribes to other components using their id values. The counterparts are mapped together by the communication translator. As soon as the required components are available, they are subscribed and connected together, and messages can be passed.

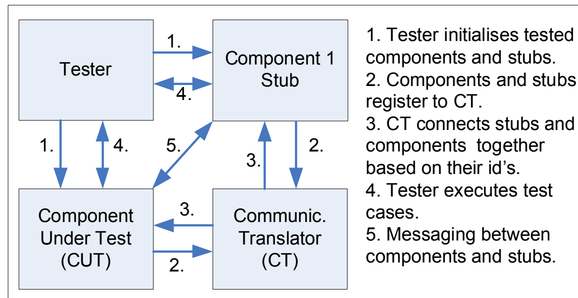


Figure 2. C2 message flow control and testing.

Using this functionality, the components are wired together to provide the test composition needed to effectively isolate and test the production components as illustrated in Figure 2. To create system compositions for testing, test components are published with the connection id of the matching production components. Thus, the system under test (SUT) sees this as a normal operating environment. To enable the creation of different system compositions for testing, the database describing the system components can also be controlled. The system configuration is set through the database, and the required components are then implemented in the test environment. These test components can then, depending on the configuration, provide test functionality such as answering with certain messages and verifying the contents and order of received messages. High reuse factor has been achieved in using parts of the test components over different test cases.

5.3 Discussion

As both C1 and C2 both work in the same domain and develop products based on the same specifications, they have many commonalities in their testing. The systems are based on a set of standard interfaces and system components that define the external structure and interfaces of the system. At the level of integration and system testing, both have mainly focused their testing on the messages between the components. Although there are differences in test implementation and system architecture, there are also similarities in the approaches taken to address the testability requirements in the internal design of the systems.

To control the system execution for testing, both take a similar approach of controlling the routing of control- and data-flow via the system messages. C1 uses the configurability of their OS message routing to enable this. C2 uses the similar functionality of their custom middleware. Both use these to enable stubbing of interfaces in their test environment, which in turn enables customized test configurations of the SUT. Both of these approaches have their own advantages. Using the services provided by the OS, there is less need to develop a custom, self-made middleware. However, using a custom-made middleware that can be used on both host and target system provides more flexibility and better possibilities for host testing, as described in section 6.

Overall, it can be concluded that for the type of testing described here, it is necessary to be able to control the messaging of the system for effective test implementation. The possibilities for this are constrained by the used SW platform.

6. Simulation Strategies

In all SW testing it is important to be able to simulate parts of the SUT for effective test implementation. As described in section 5, it must be possible to use simulated versions of components (test stubs) providing test functionality as a replacement for actual production components. However, especially in the context of embedded systems, it is also important in whole to be able to run tests in a simulated host test environment, without the need to always run on target HW [6]. This section describes the different approaches and related constraints with the interviewed companies in this regard.

6.1 Company 1

As described in section 5.1, C1 uses a COTS OS, targeted especially at embedded systems. From the simulation viewpoint, this is problematic as the OS is tied to the target HW, and cannot be used as such in a host test environment. While it does provide a separate simulation environment that enables some testing, this

simulation environment does not equal the use of actual full OS. Although C1 sees it important to effectively simulate different parts for testing, they have done only very limited host testing. Instead, they have aimed at running as much as possible of testing on target HW, where the tested component(s) and the simulated stubs are all loaded on to actual target HW. They see this to have the benefit of getting more accurate timing as well as fully matching any parameters of the target system that may affect the functionality.

In C1, the communication interface uses a common messaging protocol, built on top of standard network protocols, for all parts of the system. This is seen as an especially important enabler for building effective test automation systems, as it enables building reusable test components and services. With this type of a communication mechanism, C1 has found it possible to build multipurpose test components that can be used in testing of different parts of the system.

6.2 Company 2

As described in section 5.2, C2 uses a generic OSS OS, which can be used equally well in both a host environment and on embedded target HW. As both the target HW and the host simulation environment use the same full OS, they are a very close match to each other. Also, the same SW can be deployed both on target and in host environment without changes or visibility to the deployed SW. Only the HW interface part needs to be specific to the target system, and is typically only needed in later phases of integration. It is also possible to run parts of the system, such as HW specific startup code or databases on a different blade on a target HW, while running the rest of the system in a host environment. While the goal with testing at this level is to test as much as possible in the host environment, some of the testing such as redundancy and failover needs to be done with actual target HW, where multiple HW blades are available.

6.3 Discussion

The main difference with regards to simulation environments in C1 and C2 is in the type of OS used and the environment in which most of the testing is performed. While this is partly defined by the possibilities of the used OS, it also reflects the deeper views of the two organizations. Running as much as possible of integration testing in a host test environment is a view shared by all interviewed people in C2, whereas the opposite view is shared by the people in C1. The main reason mentioned for C1 to prefer running as much of testing as possible on target is that in this case all parameters of the actual environment are correct according to the target HW. While the optimal choice depends on many properties, both C1 and C2 recognize that testing on target HW is expensive as it requires having a large number of custom made target systems and specialized test equipment available for testing. A host testing environment also enables better control over the test environment, and the target HW issues as a whole can best be addressed in the system integration test phase, when the whole system is composed.

For C2, an effective host simulation environment is available by running their middleware and SW on top of it in a (desktop) host test environment in the same OS. As the middleware is in-house and the OS is OSS, the system supports a wide range of customization possibilities. The C1 OS based solution is tightly coupled with the internals of a third party COTS OS, which makes effective customization more difficult. Their OS is also specifically for embedded systems, and cannot be used in a host test environment as such. Instead, a specialized simulation environment is needed, which is more complex to match to target than running the actual OS. As C1 has not made much effort to use host testing, it is unclear how well this could be done.

7. Test Functionality

Enabling effective test implementation typically requires certain properties and functionality from the SUT. How SW components can support the testing process has received a lot of attention in the recent years [11]. However, these techniques are mostly considered from the viewpoint of third-party black-box components, where support is built into a single component. At a higher level, this section reviews how C1 and C2 have addressed supporting testing at a level where several components are integrated.

While the techniques described earlier for controlling messaging and using simulation are basic enablers, also more advanced functionality is needed. To effectively build automated test cases, it must be possible to observe and control different parts of the system, which when integrated can be difficult to access. This is especially true when there is a need to support testing and diagnosis of deployed products, as is the case with both C1 and C2. Finally, even when this observation and control support exists, data still needs to be made available. In embedded systems this provides its own challenges. As opposed to SW running in a desktop environment, there is often no direct visibility to the SW running on target HW, and even reading printouts from application code requires custom solutions, such as use of network protocols as used by C1 and C2. To support analysis of long-running systems, run-

time test support is also needed. In this section the functionality to support test implementation is reviewed.

7.1 Company 1

When data from an internal part of the system needs to be collected in C1, the data flow can be changed. One applied solution is using the functionality described in section 5.1; message routing is configured to go through an extra test component that collects data. A basic setting for this is illustrated in Figure 3. Similarly, by connecting various component input and output ports together, data flow in the system can be looped to come back to the sender. Using this technique requires special consideration, as the data will likely be of a different format than expected in the "abused" output path.

For direct access to deeply embedded features in C1, a technique called embedded tests is used. These are test components that are integrated into the system to provide specific test functionality. To enable this, the message routing techniques described in section 5.1 are once again used, as illustrated in Figure 4. The functionality of these components includes feeding test input data, doing comparison and transferring test result data out from internal interfaces. The embedded tests also help address other constraints such as real-time requirements and limited communication buses, by providing fast data input and processing near the interface, limiting the need for external communication. These tests are typically integrated ad-hoc where needed and not left in the system, as also the need where they are used varies. Other techniques to address performance constraints include running test data processing and transfer tasks when system load is low, by using low task priorities.

In C1 the viewpoint has been that it is difficult to provide generic observation points between different components of a system. This is both because of differences in actual implementations of components and due to difficulty to get management support for adding extra test code into the system. The priority of DFT SW development and resourcing is always put much lower than that of production SW. As most tracing then needs to be implemented on ad-hoc basis, common functionality that is used in tracing a system has been implemented in a test point library. This functionality can be integrated in different parts of the system through the library. Separate integration is needed for each test point in the system, but from thereon the library provides the functionality. The functionality of this test point library includes different functionality needed in testing, such as storing test data, moving it out of the system, doing comparisons, reporting results and monitoring system resources.

The goal with the test point library implementation is to provide functionality that makes it possible to implement embedded test and other test functionality into the system with minimal effort. This includes implementation of such functionality over different parts of a system, but also across different products in a product family. To this end, the library has been made HW platform independent. The library functionality is built on top of a HW abstraction layer (HW API), and porting the library to a new system typically needs porting the HW API code between the SW and HW, while the HW API interface stays the same.

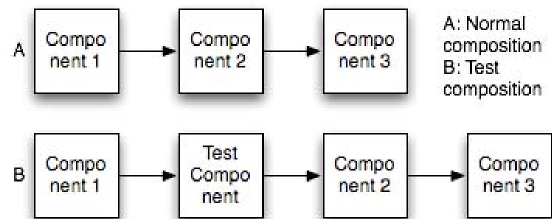


Figure 3. Embedded test component.

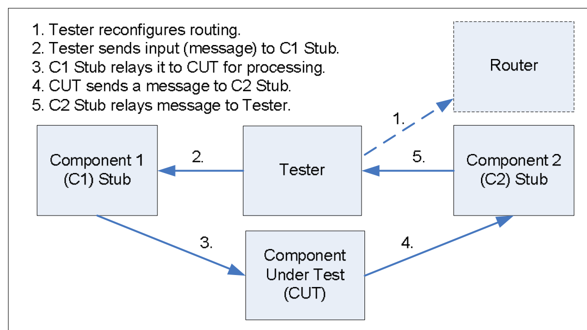


Figure 4. Embedded test.

7.2 Company 2

In C2, features to support testing and more detailed analysis of system behavior have been included in the system as first-class features. This includes both functionality to observe system behavior and built-in test functionality. While their middleware would enable using techniques such as embedded tests described in section 7.1, these have not been used extensively due to wanting to not change the SUT during testing. Instead, the goal has been to always test the system as it will be finally deployed, with no test features added only for testing and removed after testing is finished.

For systematic tracing support, development guidelines define what can and needs to be traced at different levels. Tracing is divided to different levels, and different parts of the system are defined to belong to a certain trace module. Each module contains what is

called a trace notebook to store trace data. At the finest trace levels, developers are free to put almost any trace they wish, as these levels will be compiled away when actual SW releases are made. At the higher levels, which are included in deployed products and where less trace can be produced, guidelines include always putting a certain trace level on input and output of external and internal interfaces.

The goal with the trace functionality is to always have some systematic information to work with, even if it is only possible to gather very limited amount of data due to performance constraints of deployed systems. Quality assurance policies are applied and inspections are done with experiences developers to see that all required important trace points are included in the system. In addition to storing information about system execution, all error symptoms are also logged in the notebooks. Techniques such as shared memory and stored to disk storage are used to enable access to data also in case of application crash.

The trace functionality can be configured statically through a configuration file or dynamically through a configuration interface during run-time. Trace functionality can also be set to activate or configure based on various system event triggers. Some of the basic functionality of tracing is also supported similar to the test library of C1. Mainly this is related to inserting trace statements, moving the data out of the system, data post-processing and analysis. Monitoring of system resource usage such as CPU and memory is supported and stored by the trace system.

As the more abstract levels of trace are always included in the system and can be configured dynamically during run-time, they can also be used during long testing sessions and with deployed systems in the field. Additionally, this is supported by built-in test functionality called audit tests. These tests run inside the system, when the system load is low and there are resources available to run the tests. They check the system consistency for properties such as resource leaks. Without this, it is difficult to see what has happened in the system if the system goes down after weeks of running and there is no sign of how the symptoms developed over time.

Another application of audit type tests has been in testing for errors in redundant HW blades. In this case, the system contains two or more blades that are redundant and provide fail-over functionality. During system runtime, these can be tested with known input and output data to see if they provide correct functionality. If the results are different, an error notification can be raised and the failing blade can be replaced with a correct one while the system is running.

7.3 Discussion

The approaches with the two companies with regards to including test functionality in a system have been the opposite. In C1, the viewpoint has been that no functionality to support testing are to be included in actual products, as these are not something sold to customers. As DFT support in C1 has in general been a low priority, their solutions to support this have been limited. On the other hand, the goal in C2 has been that the SUT remains unchanged during testing. From their viewpoint, it is seen that unless test functionality is a part of the actual product, the tests do not test the actual system, since the test functionality is removed in the end. Thus their supporting DFT features have been made first-class features for the system.

Due to not being able to include test functionality as a part of the actual product, C1 has developed their test point library to support testing and debugging with ad-hoc solutions. This is seen especially problematic in diagnosing deployed products and in long testing sessions. Integrating separate test support functionality requires loading a new SW version and resetting the system, which also makes the fault state disappear and impossible to debug. In general, supporting field-testing of deployed products is identified as an important area of improvement in C1. Currently, only a number of basic properties can be observed, such as number of resets inside a HW block.

As stated earlier, C2 has taken the opposite approach to fully include all test functionality in the product. This along with their audit tests provides C2 with much better support for testing and diagnosing problems of deployed products and long testing sessions. Also, as systematic tracing is included in all parts of the C2 SUT, they also share a common data format. This enables use of same tools for all parts. In C1, this has caused some problems, due to fragmentation and incompatibilities caused by the different tools and data formats taken by different organizational units.

One of the reasons for difficulties in getting support included for testing purposes into the system in C1 is often cited as people not wanting to add any (test) features that are not sold to customer, into the product. In this regard, testing has not been valued high enough to be given systematic support, but has rather been viewed as something extra to be put up with. Thus lack of management support and not valuing testing high in the company culture are some of the main reasons. On the other hand, it is not clear if better support would have been received if someone had suggested similar solutions, and argued with extended support for field testing and other cost savings. Field test support in C1 is one of the identified areas needing improvement.

8. Conclusions

This paper discussed the DFT solutions to support test automation from two companies in the European telecommunications domain, working on similar large-scale component-based embedded systems. Their techniques to support effective test automation were discussed. While the approaches taken have a lot in common, there are also a number of differences. While it is not possible to generalize from this data to all SW development and testing, a number of observations can be made that provide interesting insight into these topics. These are summed in the following:

- Testability needs to be taken into account early in the design, in the SW platform. Control over system messaging provides support for control over system execution paths and efficient implementation of test environments and configurations. A common communication protocol further provides support for implementing reusable test components.
- Especially in the case of embedded systems, a good host test environment enables efficient SW testing. When this environment matches the target system as much as possible, efficient host testing is possible. One enabler for this is using an OS that is supported on both the target HW and in a (simulated desktop) host-testing environment.
- Including supporting test functionality in the system as first-class features allows for more effective analysis of the system, including analysis of long running tests and deployed systems, and enables efficient field-testing. Effectively implementing this requires possibilities for dynamic configuration of test functionality during system run-time.
- In addition to systematic test support functionality, ad-hoc requirements are likely to arise in different points of testing and analysis lifecycle, and in this case it is useful to have support for this functionality provided in the form of a reusable library.
- Abstracting test cases from the implementation minimizes the effects of internal system changes to the test cases. This mostly applies at the system testing level, as in earlier testing phases it is often necessary to observe more detailed properties of the system.
- To make it possible to get the desired test support functionality included into the system design and to create advanced tools, management support is crucial. This requires valuing testing and system analysis high in the company culture.

9. References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd edition, Addision-Wesley, 2003.

[2] B. Baudry, Y.L. Traon, Measuring Design Testability of a UML class diagram., Information & Software Technology, vol. 47, no. 13, pp. 859-879, 2005.

[3] S. Berner., R. Weber, R.K. Keller, Observations and Lessons Learned from Automated Testing, Proc. 27th Int'l. Conf. on Software Eng. (ICSE 2005), p. 571-579, 2005.

[4] A. Bertolino, Software Testing Research: Achievements, Challenges, Dreams, Proc. Future of Software Engineering (FOSE2007), pp. 85-103, 2007.

[5] R.V. Binder, Design for Testability in Object-Oriented Systems, Communications of the ACM, vol. 37, no. 9, pp. 87-101, September 1994.

[6] B. Broekman, E. Notenboom, Testing Embedded Software, Addison Wesley, 2002.

[7] M. Bruntik, A. Deursen, An Empirical study into class testability, Journal of Systems and Software, vol. 79, no. 9, September 2006.

[8] S. Jungmayr, Identifying Test-Critical Dependencies, Proc. IEEE Int'l. Conf. on Software Maintenance, Montréal, Canada, 2002.

[9] R. Kolb, D. Muthig, Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures, Proc. of the ISSTA 2006 workshop on Role of Software Architecture for Testing and Analysis (ROSETEA2006), pp. 22-27, 2006, Portland, Maine.

[10] B. Pettichord, Design for Testability, Proc. Pacific Northwest Software Quality Conference (PNSQC2002), Oct. 2002.

[11] M.J. Rehman, F. Jabeen, A. Bertolino, A. Polini, Testing Software Components for Integration: A Survey of Issues and Techniques, Software Testing, Verification and Reliability, vol. 17, 2007, pp. 95-133.

[12] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann, 2006.

PAPER V

A Probe Framework for Monitoring Embedded Real-Time Systems

In: Proceedings of the 4th International Conference on Internet Monitoring and Protection, ICIMP 2009, Venice, Italy, 24–28 May, 2009. 7 p.
© 2009 IEEE.
Reprinted with permission from the publisher.

A Probe Framework for Monitoring Embedded Real-time Systems

Markku Pollari
Technical Research Centre of Finland, VTT
Oulu, Finland
Email: markku.pollari@vtt.fi

Teemu Kanstrén*
Technical Research Centre of Finland, VTT
Oulu, Finland
Email: teemu.kanstren@vtt.fi

***Abstract*—This paper introduces a general framework directed for system instrumentation. The introduced framework provides support for a system instrumentation approach that enables designing information capture, monitoring and analysis features into a software-intensive system. We describe the general concept, architecture and implementation of the framework and two case studies in its application. As a prototyping platform, we dealt with collecting information from Linux systems by probes created with the building blocks and interfaces provided by the framework. Overall, we demonstrate the feasibility of a more uniform instrumentation approach through this concept and its application in two case studies.**

I. Introduction

Understanding and analysing the behaviour of complex, software-intensive systems is important in many phases of their life cycle, including testing, debugging, diagnosis and optimization. In addition to these, many systems themselves are built for the sole purpose of monitoring their environmental data and reacting to relevant changes, such as detecting patterns in internet traffic. All these activities require the ability to collect information from the different parts of the system.

These basic activities and requirements in software engineering have existed since the first days of writing software. However, despite this there has been little research and activity to build support for systematic monitoring and information capture into software platforms. Instead, what is most common is the use of ad-hoc solutions to capture data where needed, as needed. In these cases, the instrumentation required to capture the information is added momentarily into the system and removed after the short-term need has passed. Recent studies still emphasized this problem, showing large-scale systems where these types of features are important but support for them is lacking [1].

In this paper we present a design concept, and its implementation and validation, for a platform to support the systematic capture and analysis of information related to the behaviour of a system and its environment. This platform is termed as the Probe Framework (PF). The PF provides support for building monitoring functionality for collecting information on the behaviour of software intensive systems and using this information for purposes such as built-in features in the software itself (as product features) and testing analysis of the systems during their development (testing and debugging) and deployment (diagnostics). The prototype implementation of PF is available as open source[1].

This paper is structured as follows. Section 2 discusses the background and motivation for the work. Section 3 describes the main concepts of the PF at a higher level. Section 4 discusses the implementation of the PF and section 5 presents the experiences from this implementation and describes two cases of utilizing it. Finally, conclusions end the paper.

II. Background & Motivation

The concept of capturing information from a system and its environment is often described as tracing the system. Similarly, in this paper we use the term tracing to describe the activity of capturing information from a running system, either with external monitoring or internal instrumentation features. The data captured is described as a trace of program execution. Many different domains make use of tracing information, such are: system security analysis, internet monitoring and protection, run-time adaptation and diagnosis, testing and debugging.[2][3][4][5][6][7]

There are various tools around for specific instrumentation and tracing on different platforms, such as DTrace for Solaris [8] and OSX [9], Linux Trace Toolkit Next Generation (LTTNG) [10] and SystemTap [11] on Linux. These tools all share a common goal to observe and store traces of system behaviour and resource use, such as CPU load, network traffic and filesystem activity. They are typically intended to provide a trace facility for the low-level resources and related behaviour of the system kernel, using solutions such as their own programming/scripting language to define where to exactly insert trace code into the operating system kernel [8][11]. For example, in our implementation of PF we make use of SystemTap, which allows one to add trace code into the Linux kernel without the need to recompile or reboot the running system [12].

These low-level frameworks provide an excellent basis for capturing low-level information from a system when ad-hoc instrumentation needs arise. However, while these tools are

*Also affiliated at Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Mekelweg 4, 2628 CD Delft, The Netherlands.

[1] http://noen.sf.net

useful for many purposes, they alone are not sufficient for efficient observation of complex system behaviour. More useful information can be gained by using advanced analysis methods such as multivariate analysis to infer additional information such as relations and similar properties from the low-level data [13]. However, what is also needed is information on a higher level, including the events, messages and interactions of different parts of the system. Also, information about the environments of these parts and their relation to the lower level details are needed.

This type of information is a part of the higher level design of a system, and it is implemented as higher-level abstractions inside the components. Thus, it is not possible to build generic components that would capture this information from all the components from the OS kernel or any custom application. When solutions such as component based middleware are used, it is possible to build part of this support into the middleware itself to capture the data [1]. However, for an effective and descriptive trace, application specific tracing is also needed. For this level of tracing several frameworks exist, such as Log4J [14] and syslog [15]. Additionally, when the availability of such features and information is highly valued, customized support for these have been built into the system as first-class features [1].

The above descriptions show how effective analysis of software intensive systems requires many different types of traces to be supported, collected and analysed together. Different tools need to be used effectively in different steps and finally combined as one for both built-in features and external analysis. Only in this way is it possible to provide the required support to get a definite view of the behaviour of a complex system.

From this viewpoint of complete system analysis and its support through the life cycle, the described trace tools and frameworks suffer from a set of issues. The tools use their own interfaces, custom data formats and storage mechanisms. Additionally, often the storage is only considered in the form of a local filesystem with the intention of being manually exported to external analysis tools or read as such by humans. Simply accessing this information from an embedded system can be very difficult as these systems are often limited in their external interfaces. Even where this is possible, in the case of a deployed system, it is not always cost-effective for someone to go to the field site to examine the trace file. Additionally, the trend for relying on ad-hoc temporary tracing solutions makes it very difficult to capture a meaningful trace of a system as there is no built-in support to be used when needed. The lack of design support for proper tracing from the beginning further brings problems such as probe effects, where addition of temporary trace mechanisms changes the timings of the actual running system that is to be analysed [16].

To address these issues, to build a basis for effective system level tracing, analysis and related program functions, we have developed a trace platform called Probe Framework (PF). Our prototype implementation is created on Linux and enables the collection of trace information both from kernel and user space probes, through a single unified component in the system. By starting with the goal of building these features into the system as first-class features we make it possible to address properties such as probe effects, information access, limited resources and real-time requirements. With a commonly shared and customizable format for the collected trace, it is possible to store and export this information to different analysis tools. With unified interfaces inside the platform it is also possible to easily design built-in features that make use of information from all the various tools. As the main intent is to build a higher-level abstraction mechanism, we use existing tools such as SystemTap and integrate it to the PF. The PF and its main concepts are described in more detail in the following sections.

III. General concept

On a higher level, the PF is a part of a larger concept which includes three main components. The PF provides the needed support as a platform to capture the trace information from the system under test. An information database server is used to collect the trace information and provide the means to query, filter and export the trace to analysis tools. Various trace analysis tools can be used to analyse the information provided. This includes tools specifically for trace analysis and also tools more generally intended for analysis of data, such as multivariate analysis. For example, experiences on using a multivariate analysis tool to analyse the network functionality and behaviour of a system have been studied in [13]. In addition to making use of the captured information in external analysis tools, it is also possible to make use of it as part of built-in product features for processes such as adaptation, testing and analysis. This overall architecture is described in figure 1.

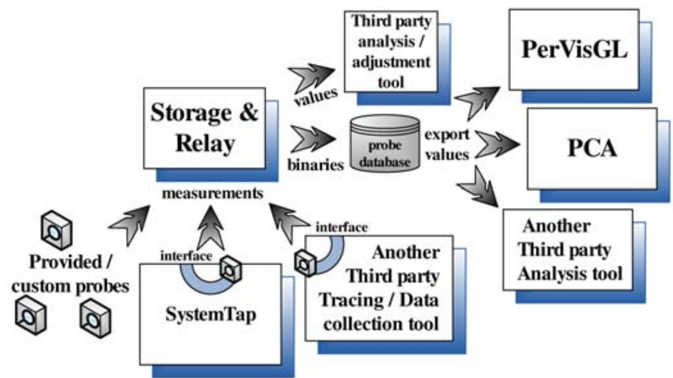


Fig. 1. High level PF collaboration

The Probe Framework itself has a layered architecture as presented by Buschmann et al. [17]. The PF's architecture is divided in three main layers; Basic services, monitoring services and test services. The term probe, in the context of this paper, means the entity that is formed by utilizing the different service layers to create the functionality for collecting and handling the monitoring of some aspect of the target system. Each layer builds on the functionality of the layers below it, as described in figure 2.

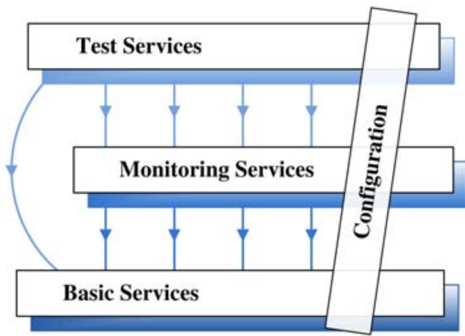


Fig. 2. Layered architecture

The basic services contain services deemed necessary for information handling, such as data buffering, storage and relaying to external database. The basic services are general for all the probes, and offer the support for fast implementation of the upper level services. The basic services comprises of three parts; first part is the probe interfaces, second is the binary formatter and the third is the communication handler. These are illustrated in figure 3.

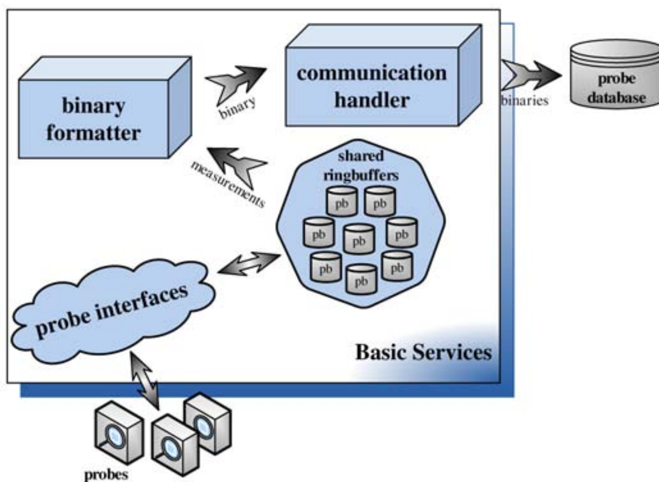


Fig. 3. Structure of basic services

Together these parts take care of all the data management of the tracing as described in figure 1.

The monitoring services offers a set of readily provided interfaces and probes to attach to the basic services. The actual services at this layer are used to capture and monitor different values, such as memory consumption and CPU usage, and their evolution in the system. Many of these basic monitoring services are provided as ready probe components in the implementation of the PF, including CPU load, memory consumption and network traffic monitoring. Further, they provide simple interfaces for building new monitoring services on top of them without the need to concern with the complex internal details of the data management.

The top layer, test services, is the most implementation dependent and is where the system specific functionality can be build. It relys on using the basic services and monitoring services. For example, functionality can be built to inject test data into the system, use a provided set of monitors to see how the system behaves and store the test results using the basic services. Similarly, in a running system the same monitors could be used from a test service (or more accurately, built-in functionality) that adapts the system's runtime behaviour and use of components based on thresholds set for monitored values such as memory consumption, CPU load and network traffic patterns.

IV. Implementation

The main implementation platform here is the embedded real-time Linux systems. This platform was chosen as it provides an interesting and realistic platform for the implementation of this type of software, with both possible issues and available options. These issues include the strict timing requirements and limited resources inherent to the embedded real-time systems. Yet, even as we are dealing with embedded software where we know all the running software beforehand it needs to be possible to access the whole platform including the kernel. With Linux as the operating system, this is particularly easy as the whole operating system (OS) is open source software (OSS). Additionally, the PF's basic services of data storage and transfer have also been implemented in Java. However, this implementation and platform are more limited and are thus only discussed where it provides insight into the differences between the implementations on different types of systems.

Although conceptually one, the actual implementation of the probe interfaces in basic service layer is divided in two. The major reason for this is the way execution in operating systems typically takes place, in either user space or kernel space. This separation also acts as a divisor for the probe types, resulting in a split between kernel probes and user probes. Additionally, in Linux as well as most modern OS's each user space process runs in its own virtual memory space, and thus cannot normally access the memory of other processes nor can other processes access its memory [18]. However, for effective implementation of the PF, all the trace data for a single system needs to be centrally managed. This requires that there needs to be a single component that takes care of the data management for all the probes deployed, either in kernel or user space. This division of implementations, probes and interfaces is described in figure 4.

There are basically two fast enough ways to address the data relay requirements, one is that the processes can request the kernel to map a part of another process's memory space to its own, and the other is that a process can request a shared memory region with another process. These shared memory regions are also useable between kernel space and user space processes. The choice made when developing the probe framework was in favour of the shared memory as it works both in kernel space and user space. The shared memory is used in both between kernel space and user space

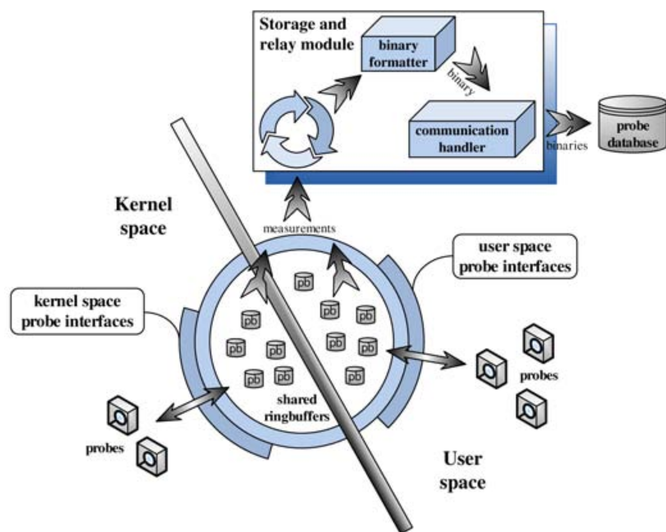


Fig. 4. Division of probe interfaces

and between processes in user space, such that a single data management component takes care of the basic services.

The storage and relay module resides in the user space, conforming to a general guideline for operating systems [18]; perform actions in the user space if possible, as kernel space should be reserved for parts that absolutely must be there as they require special privileges. Kernel code can also crash the whole system with its privileges and thus these parts need to be absolutely secure and reliable. Since we do not need to perform actions with special privileges it makes sense to locate most of the code in the user space. This is also one of the main reasons for why the shared memory regions are used between kernel space and user space, and also inside user space. This was all in order to separate the trace handling functionality from the probes and to centralize the trace collected by the probes. This enables the storage and relay module to access the trace, format it and provide it for higher layer functionality or simply relay it to the end storage as requested. All this reduces the interference induced to the target by the monitoring activity conducted by the probe as all the ”extra” processing can be done separate to the probe in its own process. Another benefit for having the storage and relay module, i.e. the basic services, in its own process in user space is that it enables easier configuration of the provided services.

The binary formatter part of the said module is the simplest part of the component; it is as the name suggests a formatter used in changing the collected data to a more manageable form. The reason for the use of binary format is to provide an effective, single format to share the data between different tools, layers and databases. The intent is to support probes created in different programming languages, running on different platforms and with strict constraints on memory and real-time requirements typical to embedded systems. Implementing this effectively is not trivial; however, the user is completely shielded from the details by the provided abstraction interfaces. The communication handler is the second part of the storage and relay module. This part handles the data transfer to end storage locations, takes care of the configuration of the storage and relay module and manages the data extraction from the probe buffers.

In practice the basic services are implemented as a shared library component, meaning that the implementation code needs only a single instance (code segment) to reside in the memory during runtime. This makes synchronizing all the trace data for a system overall much simpler due to only having a single instance of basic services for a system at any time. The library is implemented as a dynamically linked library, which is linked to the components during execution, meaning it is shared also between different processes in the user space. For the kernel space there is a similar component.

Configuration

In order to cope with a variety of different devices, the configuration possibilities of the probe framework are substantial. Each probe can be configured separately, as can be the storage and relay module. Various possibilities for accumulating for the different capabilities of the target system are offered by the probe framework. The output possibilities and replacement strategies, etc. used by the storage and relay module are all configurable to suit the system’s capabilities.

The major control features of the probe framework reside inside a configuration file that is read during the activation of the storage and relay module. This allows configuring the basic parameters such as overall buffers, general policies and storage mechanisms externally. Another layer of control is embedded in the creation of the probes, during the implementation of a probe the creator can use custom settings to define probe specific values or leave them out in which case the default generic values will be used. The probe specific attributes include buffer size, preferred storage location, priority, timing accuracy and presumed output type. Additionally, the creation of output types used by the probe introduces control as it is possible to use prioritized data types for increasing the probability that the collected trace reaches its storage location. In order to address restrictions such as keeping the monitoring overhead low, several policy parameters can be defined. One is the possibility of discarding parts of the collected trace if the basic services cannot run fast enough to relay it to a storage destination. This is further influenced by the priority set to the trace through the configuration. More advanced policies can be implemented inside custom probes, such as sampling or time-interval captures.

Instrumentation

As described earlier, instrumentation is divided into two main types of probes: kernel and user space probes. A distinction can also be made between internal and external instrumentation. Internal refers to embedding the instrumentation code to the software object that is part of the monitored system. In this case the probe is an integral part of the program code. External instrumentation refers to the probes where no modifications are made to the system software itself. Instead

a stand-alone process handles the monitoring from outside of the target software. The PF provides support for all these different types of instrumentation. Custom kernel and user space probes and built-in functionality can be created using the services provided at the different layers of the PF. A set of external instrumentation components are provided as kernel probes and processes to collect and analyse generic properties such as task-switches, CPU load and network traffic. More such custom components can also be easily created. All these instrumentation possibilities share the set of basic services that remain unchanged between different implementation possibilities. Therefore, it is simple to analyse the collected trace data, build additional functionality or make other use of the instrumentation data from all different probes and monitoring tools through the provided interfaces.

V. EXPERIMENTS & EXPERIENCES

To perform evaluation of the PF concept, its implementation and application we carried out two case studies. Both of these are in the domain of monitoring embedded software-intensive systems. This means we focused on using the monitoring services layer of the PF, and indirectly the basic services through the monitoring layer. In a sense our implementation is also part of the test services layer, as we built custom functionality to use the lower layers. We start with describing each experiment and the overhead cost their implementation had on the system we were analysing. We then describe our experiences in using PF as a platform for implementing overall instrumentation for system monitoring.

The two case studies we have performed are monitoring kernel task switching and the memory usage of different processes. The memory use monitoring case was conducted on an embedded system that was provided by Espotel[2]. This platform, called Jive[3], is a battery-powered, touch screen equipped PDA type of a device with broad connection interfaces. For the task switching instrumentation a typical desktop PC was used.

Task swith case study

The task switching case study focused on the scheduling of processes (tasks). In a typical modern OS there are numerous processes running at the same time [18], and the scheduler handles the execution of tasks by dividing the CPU resources to slots and distributing these slots to the tasks. Our goal was to build a monitoring probe to capture the information on how the task switching is performed with the given usage scenarios. The visibility of the scheduling activity in this scenario is strictly for the kernel space only, and as such, the monitoring had to be implemented as a kernel probe. For this case study, we collected three types of events:

- Task activate
- Schedule
- Task deactivate

[2] http://www.espotel.fi
[3] http://www.espotel.fi/ratkaisut_jive.htm

The schedule event means that the running task is switched to another, the meanings of activate and deactivate are a bit more complex. The activate and deactivate denote that the task is moved to or away from the run queue. For simplicity it can be tought that these two events tell when the task can be run i.e. scheduled. The instrumentation used in this case is an in-line probe in the kernel's scheduler, implemented via SystemTap. The code that uses the PF's probe interfaces is added to the SystemTap probe script as embedded C. Similarly, other existing monitoring applications could be integrated to the PF by using the provided probe interfaces, see figure 1.

As the instrumentation is done using external instrumentation it also serves to provide a generic reusable kernel monitoring probe for future use when task switching needs to be analysed. This is illustrated in figure 5.

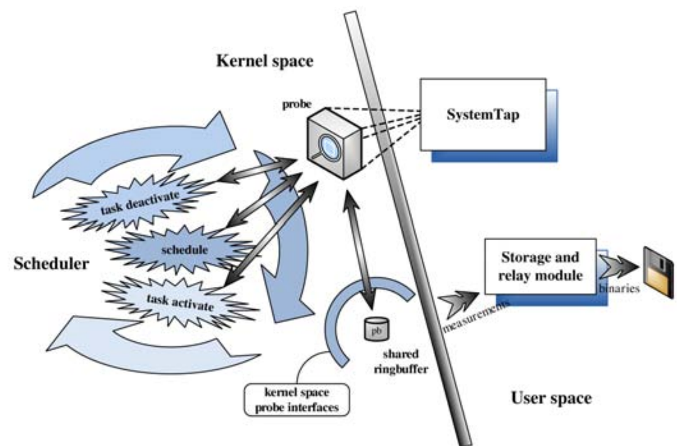


Fig. 5. Task switch instrumentation.

In this case, as task scheduling happens numerous times each second, the used instrumentation is extremely intrusive. There are bound to be consequences due to the instrumentation code. As we want an accurate picture of the task scheduling, all the events need to be collected and no sampling can be used. Therefore, the overhead is so high that the probe is only useable temporarily for purposes such as diagnostics or to provide basis for performance analysis. In this case we do not have any hard real-time requirements so the temporary inclusion of the probe and the temporal effect it poses on the execution is acceptable. Due to the use of SystemTap this probe can also be enabled (included) temporarily in a running system and disabled (removed) when the required diagnostics data is collected. Thus, it shows how it is possible to create PF probes that can still be included in the system probe set also in deployed systems while they are only used in ad-hoc style during system execution. Concerning the analysis results, it needs to be taken into consideration that the probe code will consume part of the CPU time, causing skew in the time interval trace, and that the storage and relay module that runs in the user space as a normal task will appear on the obtained task switch trace.

In our case study, we used the obtained trace for different

purposes, including the characterization of the process load running on the system as described in [19], for analysing task blocking and scheduler performance. The overhead of the PF was measured by capturing system timestamps as jiffies, the jiffies describe system time/clock ticks as 4ms intervals. The stamps were collected at the beginning of the instrumentation and at its end. To obtain a reference point, the duration of the instrumentation was measured, and then the same captures were done in a system without the instrumentation, using the measured duration as a time interval between the captures. To give a better picture of the effect the instrumentation had, the overhead is given as the reduction caused to the true idle time of the target system. The overhead is calculated with the following formula:

$$\frac{\left[\frac{DI}{J} - (CJE - CJB)\right] - \left[\frac{DI}{J} - (CJEI - CJBI)\right]}{\frac{DI}{J} - (CJE - CJB)} * 100\%$$

DI = Duration of the instrumentation,
J = Duration of a jiffy,
$CJEI$ = Captured jiffies at the end of instrumentation,
$CJBI$ = Captured jiffies at the beginning of instrumentation,
CJE = Captured jiffies at the end of idle,
CJB = Captured jiffies at the beginning of idle.

The calculated overhead for this case was 16%. Overall, this could be considered a high cost for instrumentation. However, for an analysis case where the monitoring instrumentation is very intrusive i.e. in one of the most frequently executed function of the kernel, we do not consider this to be a bad result at all. This probe is only intended to be used as a temporary analysis aid and not as a fully included production class feature.

Memory case study

In the memory usage case study the focus was on user space instrumentation. The instrumentation target was the procfs, process information pseudo filesystem, which is an interface to the kernel data structures and provides information about the processes on the system. This case study is illustrated in figure 6.

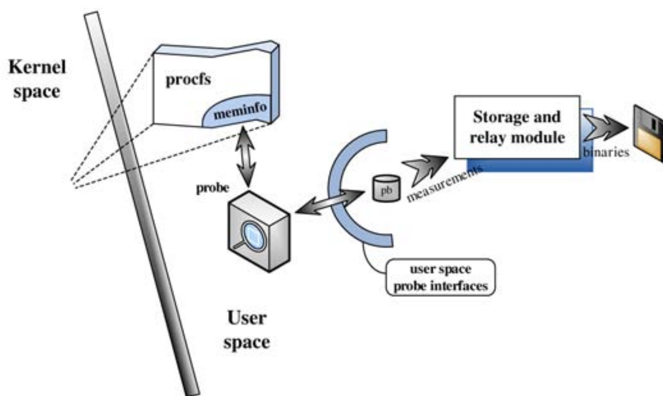


Fig. 6. Memory usage instrumentation.

The targets of interest here are two memory usage illustrating attributes:

- Total amount of used physical memory
- Total amount of used (memory) swap space

In this case we have more options to control how we wish to implement the instrumentation. The probe was implemented as an external probe that functions as a normal task in the system. The probe was set to sample the procfs for memory information every three seconds. This is adequate as our goal was to observe how memory consumption develops over time. Our intent was to use this information for observing memory leaks and growth of consumption over time, similar to audit tests inside a running system as described in [1]. This type of memory problems are considered to be among the most common and difficult to debug due to the long time they take to develop [20]. Thus this type of information is useful to have available to describe the development of the symptoms and to analyse their cause over time.

For this case, the overhead caused by the used instrumentation was measured using the same method as in the previous case. The induced overhead was 2%, which is not overly much and could still be further improved with more efficient integration with kernel functionality.

Generic notes

After our trials with the two case studies, we can say that the framework provides a reliable and efficient instrumentation interface with high potential for reuse. We implemented two highly reusable and generic probes. Due to their very generic nature, they can be reused as is or with minor modifications in other contexts. As a generic framework is also bound to be used more frequently and by more people and projects, the code will also become more reliable and optimized over time than separate custom solutions. That is, the more the PF is used the better it becomes. This makes it more likely that found problems are in the system itself and not in the instrumentation code.

As noted earlier, the basic services of the PF have also been implemented on the Java platform. As the PF's implementations both share the same file formats and protocols, we have also been able to successfully use them together. In this sense, through the shared information database storage and export facilities it is possible to get a view of systems with varying component implementations. It is our experience also from these implementations that the simplicity of the provided interfaces is a key to their easy adoption. They must be simple and easy to use and not get in the way of the developer. By hiding all the complexity of trace storage, processing and access behind simple interfaces the PF becomes also more convenient to use. And, that is what the PF aims for, to be a general reuseable approach for instrumentation that lets the developers better focus their efforts on implementing the actual product rather than spend overly much time on creating ad-hoc instrumentation solutions.

Regarding the usability of the Probe Framework, it is not limited to the context of embedded real-time systems. Those

attributes are merely something that create a challenge for the PF i.e. limit the available resource etc., and in no way limit the environment that the PF concept is suitable to. Similarly, the prototype implemetation being Linux specific doesn't indicate that the PF concept couldn't be used in a different OS. The PF's mentioned Java platform implementation for instance is not limited to the Linux environment.

VI. Conclusion

The probe framework described in this paper provides the means to build and later on reuse system instrumentation approaches effectively and reliably. It provides support for the basic requirements of storing and accessing data, as for more advanced needs processing, monitoring and building new functionality to use the traced information is supported by the PF's higher layers. Two cases studies where the PF was used were carried out to validate the different uses of the framework. These cases used the provided building blocks and interfaces to build generic, reusable probes for important system information.

The nature of embedded systems is that there is little consistency between different devices, having led to creation of customized solutions for information access. Here, we have shown that for a system where the PF is available it provides a basis for a uniform instrumentation solution. Generic probes can be reused across systems and new ones implemented by using the provided building blocks and interfaces. The reuse of the framework and probes thus leads to reduction of the implementation effort and also to increased reliability as the found problems are more likely to be in the system itself and not in the instrumentation code.

For easing the lifespan testing/diagnostics/management of the target system the probe framework can be very useful. Given that the probe framework is intended to remain in the target system after deployment, it can provide its services during the targets lifespan. Therefore, it can hasten the detection of the possible problems and offer the testing services and monitoring services during the targets lifespan. In practice, the probe framework requires the shared library, storage and relay module and the various probes to remain in the target, to provide its services after the target has been deployed. Overall, the space requirement of the probe framework is minimal, but naturally its presence will affect the rest of the system and needs to be considered.

For further details on the probe framework tool and instrumentation methods [21] offers an in depth view.

References

[1] T. Kanstrén, *A Study on Design for Testability in Component Based Embedded Software*, SERA 2008.

[2] A. Hussain, G. Bartlett, Y. Pryadkin, J. Heidemann, C. Papadopoulos and J. Bannister 2005. *Experiences with a continuous network tracing infrastructure*. In Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data, Philadelphia, Pennsylvania, USA, August 26 - 26, 2005.

[3] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song and J. Xu *HMTT: a platform independent full-system memory trace monitoring system*. In Proceedings of the 2008 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems, Annapolis, MD, USA, June 02 - 06, 2008

[4] K. Yamanishi and Y. Maruyama, *Dynamic syslog mining for network failure monitoring*. In Proceedings of the Eleventh ACM SIGKDD international Conference on Knowledge Discovery in Data Mining, Chicago, Illinois, USA, August 21 - 24, 2005.

[5] M. Diep, M. Cohen and S. Elbaum (2006) *Probe Distribution Techniques to Profile Events in Deployed Software*. In: ISSRE 06: Proceedings of the 17th International Symposium on Software Reliability Engineering, IEEE Computer Society, Raleigh, NC, USA, pp. 331-342.

[6] H. Giese and S. Henkler (2006) *Architecture-Driven Platform Independent Deterministic Replay for Distributed Hard Real-Time Systems*. In: ROSATEA 06: Proceedings of the ISSTA workshop on role of software architecture for testing and analysis, ACM, Portland, Maine, USA, pp. 28-38.

[7] S. Elbaum and M. Diep (2005) *Profiling Deployed Software: Assessing Strategies and Testing Opportunities*. IEEE Transactions on Software Engineering 31, pp. 312-327.

[8] B. M. Cantrill, M. W. Shapiro and A. H. Leventhal, *Dynamic Instrumentation of Production Systems*, USENIX annual technical conference, Boston, MA, 2004.

[9] Mac OS X. Website, [17.12.2008] http://www.apple.com/macosx/

[10] M. Desnoyers and M. Dagenais, *LTTng: Tracing across execution layers, from the Hypervisor to user-space*, Ottawa Linux Symposium 2008.

[11] F. Eigler, *Problem solving with systemtap*, Ottawa Linux Symposium 2006.

[12] SystemTap. Website, [08.12.2008] http://sourceware.org/ systemtap/

[13] P. Tuuttila and T. Kanstrén, *Experiences in Using Principal Component Analysis for Testing and Analysing Complex System Behaviour*, ICSSEA 2008.

[14] Apache log4j. Website, [17.12.2008] http://logging.apache.org/log4j/

[15] C. Lonvick, *The BSD Syslog Protocol*, RFC Editor, 2001.

[16] H. Thane and H. Hansson (1999) *Handling Interrupts in Testing of Distributed Real-Time Systems*. In: RTCSA 99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society, Washington, DC, USA, p. 450.

[17] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996) *Pattern- Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, New York, NY, USA, 476 p.

[18] A.S. Tanenbaum *Modern Operating Systems* 3rd edition, Prentice Hall, 2008, 1104pp.

[19] M. Jaakola (2008) *Performance Simulation of Multi-processor Systems based on Load Reallocation*. Masters thesis, Oulu University, Department of Electrical and Information Engineering, Oulu.

[20] J. Vincent, G. King, P. Lay and J. Kinghorn (2002) *Principles of built-in-test for run-time-testability in component-based software systems*. Software Quality Control 10, pp. 115-133.

[21] M. Pollari (2009) *A Software Framework for Improving the Testability of Embedded Real-time Systems*. Masters thesis, Oulu University, Department of Electrical and Information Engineering, Oulu.

PAPER VI

Observation Based Modeling for Model-Based Testing

In: Submitted to the Journal of Software Testing, Verification and Reliability, 2009.
Reprinted with permission from the publisher.

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2007; **00**:1–7 (DOI: 10.1002/000)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/000

Observation Based Modeling for Model-Based Testing

Teemu Kanstrén[1,*], Éric Piel[2], Hans-Gerhard Gross[2]

[1] *VTT*
Kaitoväylä 1, 90571 Oulu, Finland
teemu.kanstren@vtt.fi
[2] *Software Engineering Research Group*
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{e.a.b.piel,h.g.gross}@tudelft.nl

SUMMARY

Model-based testing represents a powerful means for automated generation of test cases. However, creating a useful model requires expertise in the (formal) modeling languages of the tools used, and experience with the general concepts of modeling for model-based testing, in order to achieve effective test generation. These requirements make the adoption of model-based testing difficult and costly. An efficient approach to ease these requirements is by generating an initial model based on observations made from software execution and using this as an advanced starting point for model-based software testing and verification.

This article presents such an approach. Its contributions are a novel technique to generate an initial model out of observations, suitable for model-based testing, and a method supporting and guiding its application to software testing, and verification. Both the model generation and the presented method for its use are evaluated through application to a concrete sub-system in the safety/surveillance domain. The study shows, that a suitable initial model can be generated automatically, and further refined by the user following the method, for system verification and testing. The study demonstrates how residual defects and specification inconsistencies can be detected. Copyright © 2007 John Wiley & Sons, Ltd.

Received 30 August 2009

KEY WORDS: Model-Based Testing, Observation Based Modeling, Software Testing, Verification

*Correspondence to: VTT
Kaitoväylä 1, 90571 Oulu, Finland

Contract/grant sponsor: This work has been supported by the Nokia Foundation
Contract/grant sponsor: The work presented in this paper has been carried out partially under the Poseidon project in cooperation with the Embedded Systems Institute (ESI), Eindhoven, The Netherlands, and supported by the Dutch Ministry of Economic Affairs; contract/grant number: BSIK03021

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls *[Version: 2007/09/24 v1.00]*

1. INTRODUCTION

Testing is the most commonly used approach in industry for verification and validation of software, and it can be regarded as the ultimate review of a system's specification, design, and implementation [12]. Model-Based Testing (MBT) refers to automated test case generation techniques based on formalized descriptions of the system under test (SUT), in contrast to hand-crafting test cases from other available (non-formal) documents, or the source code [33]. Since testing can, often, consume up to half of the overall development cost for a software project, while it adds nothing in terms of functionality to the software, there is a strong incentive towards test automation with MBT. However, creating a useful model for MBT requires expertise and experience with the general concepts of modeling and the used MBT tool notations, in order to achieve effective test generation.

An effective approach for supporting the difficult and costly behavioral model design and construction process is to generate a (partial) model out of captured observations automatically (e.g. [13, 2]). This approach is termed here as *observation-based modeling* (OBM). Obviously, this method can only be "boot-strapped" from existing runtime scenarios and their executions (such as field data and unit tests [7]). From the test automation perspective special consideration is needed when using models automatically generated from captured observations. Such models describe the observed *actual* behaviour of the SUT instead of its *expected* behaviour according to a specification. As such, human involvement is needed when using these generated models as no computer program can automatically know what is the correct *expected* behaviour of any given other program.

Although many techniques have been proposed for mining behavioral models for software systems, the produced models are not suitable for use with MBT tools, and the techniques do not provide guidance for using them in the context of MBT. Instead in MBT, the approach has been to create test models manually, based on system specifications. The contribution of this article are

- a novel technique for automatic generation of an initial behavioural model, suitable for MBT, based on observations captured from SUT behaviour (execution traces),
- a method to support the use of these models for testing and verification of the SUT with the help of a MBT tool,
- an implementation of the approach for Java based components, and
- an evaluation of the approach with a case study, including lessons learned.

The generated model provides support for the modeling process, letting the user start from an advanced initial model suitable for MBT. The presented method provides support for the MBT user to turn the initial model into a complete description of the SUT, while continuously verifying the correctness of the implementation against its specifications with the help of the MBT tool. The approaches presented are evaluated through their case study application in a real vessel traffic surveillance system for which initial system behavioral models are devised automatically, refined and verified manually, while continuously using a MBT tool to execute the model and generate tests.

The article is structured as follows. Sect. 2 briefly outlines related work on the techniques relevant to this article, and summarizes our contributions. Sect. 3 describes the tools and algorithms used for model generation. Sect. 4 describes the method for using the generated models as starting point for model refinement, verification and testing. Sect. 5 summarizes a concrete application of OBM for testing part of a maritime surveillance system, and discusses experiences with the techniques. Finally, Sect. 6 summarizes and concludes the paper with directions for future research.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

2. BACKGROUND AND RELATED WORK

This section describes the concepts relevant to the topics discussed in this paper, presents existing work on which we build, and relates the approach presented to existing literature. The contributions of this article vs. the related work are summarized in Table I.

2.1. Background

The term Model-Based Testing has many definitions. We follow Utting and Legeard [33] who describe MBT as "Generation of test cases with oracles from a behavioural model". The model describes the expected behaviour of the SUT, and is used by a MBT tool in order to generate test cases, in the form of method invocations sequences plus input data. Validation of the correctness of the responses from the SUT is realized by test oracles that check the expected output data and interaction sequences. Test oracles are also typically built into the model.

In the traditional approach to MBT, (non-formal) specifications of the SUT are manually transformed into suitable (formal) descriptions for machine processing. A MBT tool is then used to generate test cases from the formal descriptions which are executed on the SUT in order to validate its observed behaviour against its specified behaviour.

Two types of models are relevant to our work. First, finite state machines (FSM), and Extended FSM (EFSM), are commonly used for behavioural modeling and model-based testing [33]. Both describe the system in terms of control states and transitions between these states. States are externally visible abstract representations of a system's internal variable combinations. They are modified by the effects of transitions, and initiated through stimuli sent to the SUT. EFSM models add conditions to the FSM representation in order to define explicit conditions for triggering the transitions.

Second, dynamic invariants (models) are commonly used in dynamic analysis of software behaviour. In general, dynamic invariants can be described as properties that hold at certain points during execution of an SUT [11]. Therefore it is important to note that they do not generalize to describe all possible behaviour of the SUT. Example invariants are `x > 1`, stating that in all observed executions the value of x was always greater than 1, or `x always in y[]`, representing the fact that the observed value of x at any time during the observed executions is included in the values stored in the array `y`. Support for inferring these invariants from a set of program executions has been implemented in a tool called Daikon[†].

OBM as presented in this article is based on dynamic analysis as underlying model extraction technique, and it shares the basic properties of reverse engineering and program comprehension [25, 26], also domains relying on dynamic analysis. The specific goal of OBM is to use the reverse-engineered models for testing and verification, and analysing and understanding them is important for their effective use. Supporting this process from the program comprehension viewpoint has been described in detail in [17].

A commonly used technique in dynamic analysis is the tracing of method and function calls and using the data obtained for program comprehension and modeling [14]. Most modern programming

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

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

Softw. Test. Verif. Reliab. 2007; **00**:1–7
DOI: 10.1002/stvr

Table I. Summary of our contribution vs. related work.

Topic	Existing work	Added contribution
Test process	Test generation from a state-based model with a tool (MBT, [33]). Conceptional layout of a OBM approach without implementation and evaluation [4]. Application of generated model in regression testing [9, 28, 8].	Integration and realization of techniques and tools for automated generation of a complete EFSM for verification and testing. Provide a practical implementation, algorithms, integration of tool support, evaluation, plus method with guidelines, best practices and lessons learned.
Model generation for testing	Application of generated input and captured invariants as a model for proposing new tests [22]. Approach to automatically turn these invariants into unit test checks for data values (Agitator[5], Eclat[27], Xie and Notkin [35]) Generation of an EFSM to be used for selecting tests from an existing test suite (no model code suitable for MBT) [21]).	Generation of an EFSM to be used for generating new tests (with MBT tool), and its use for verification of implementation against specification. Provide means to visually check the EFSM against the (non-formal) specification, plus generation and checking of interaction sequences, and data values.
Test oracle generation	Implementations to check that trace properties [28] or invariants [8] hold as regression tests. Checking of exceptions [8] or user Interface error codes [24]. Support for user provided test oracles [24].	Abstract complete interaction and return value oracles for the EFSM from the traces. Provide test oracles for all generated tests. Generation of application specific oracles automatically.
Mock obj. generation	Generation of mock objects for specific focused unit tests [29, 31].	Generate of mock objects for a MBT model, usable for generating a number of tests.
Input data	Usage of serialized objects[27], captured invariant values[5], random and other data factories [5] as inputs.	Application of invariants and random data factories in MBT model generation.

languages have methods as basic building blocks, so these techniques can be applied extensively. In addition, external tracing mechanisms, such as aspects [18], can be applied without having to modify the source code. Various tools provide models and visualizations from the trace data, e.g., as sequence and scenario diagrams, graphs and custom visualizations [14].

2.2. Related Work

The usefulness of possibly deriving models for MBT, based on observations captured from execution scenarios, has been discussed before by Bertolino et al. [4]. However, they did not take their approach beyond conceptional discussion. In this article, we present a practical implementation of these concepts

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

including model generation, guidelines for the use of these models for SW testing and verification, and a practical evaluation with an experimental study and lessons learned.

Apart from the overall approach, there are a number of techniques related to the work presented here. Both Ducasse et al. [9] and De Roover et al. [28] use logic-based queries on SUT execution traces to test legacy systems. To validate the assumptions about the SUT, they use logic queries on the traces and define a set of trace-based logic testing patterns. Both approaches use queries as tests for possible software regressions after updates, and for supporting the understanding of a program, by creating and validating assumptions about it. We provide a model based on similar traces, and apply it to MBT.

Whereas our focus is on dynamic analysis, related tools and techniques also exist in the field of static analysis. For example, Walkinshaw et al. [34] use symbolic execution to derive state-machines from source code, including the paths that lead to these transitions. They describe how they support inspections and program comprehension. Our focus on MBT but the generated models can also be used for these activities.

D'Amorim et al. [8] apply symbolic execution and random sequence generation for deriving method invocation sequences in order to generate unit tests. Their test oracle checks for yet uncaught exceptions, plus monitors the results of executions for violations of an operation profile described by Daikon invariants over all processed variables and values. We also generate test oracles as part of the model, including verification for interaction sequences and checking of captured invariants for return values, in a format suitable for MBT.

Tillmann and Schulte [31], and Saff et al. [29] provide means to automatically generate mock objects for the SUT. Tillmann and Schulte use static analysis (symbolic execution) and Saff et al. use dynamic analysis to capture the behaviour expectations and return values for the mock objects. Both focus on one test at a time, to allow the generation of mock objects for exactly the purposes of this test. The test for which mock objects are defined is determined by factoring a larger test to smaller tests [29], or based on static analysis of code with symbolic execution [31]. We generate mock objects in a similar fashion but usable for all tests generated with the MBT tool.

Lorenzoli et al. [21] present an algorithm called "GK-tail" used to generate an EFSM from execution traces from FSM and Daikon-invariants, which is similar to our approach. The EFSM is used for test case selection and for building an optimal test suite from existing test cases in order to optimize coverage of the model. In other work, they also compare the interactions of a component deployed in a new context in order to observe changed behaviour [22]. Our approach uses similar building blocks for generation of the EFSM, and it shares the application domain of test automation. However, the model representation and the application are different from [21]. They do not generate tests from the model, and they do not use the model as an executable specification for verification.

Lo et al. [19] mine temporal rules (invariants) from captured observations. These take the form of premise and consequence, where the premise is noted to be followed by a consequence over time. This has similarities with our use of interaction sequences as a basis for an initial FSM used as input for our EFSM generation algorithms. However, Lo et al. [19] focus on this subset only, while we further generate a complete EFSM suitable for MBT.

Finally, Mesbah and van Deursen [24] build an FSM for web-application user interfaces with the help of an automated crawler tool that is used to exercise the user interface and capture interaction sequences that cause changes in the interface's document object model (DOM) tree representation. A change in the DOM tree constitutes a new state, and this information is used to model the FSM. Transitions are the clicks (input) to the SUT that caused these changes in the DOM tree. They use a

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

set of their own invariants specifically built for web-applications to describe the expected changes in the DOM tree in response to input as test oracles. We use an FSM and invariants as a basis for our model generation, as well. However, our generated models are different. We target specifically MBT and generate the an executable EFSM usable for SW testing and verification with the help of a MBT tool. We also generate more specific oracles from the traces, whereas their focus is more on generic and user defined oracles.

3. MODEL GENERATION

This section describes our approach of generating the initial model to be used for MBT. We use a number of tools to support our approach: ModelJUnit[‡], a MBT tool using Java as a modeling notation, JUnit[§], a unit testing framework for Java, Daikon, ProM[¶], a process mining tool, and EasyMock[‖], a mock object framework.

The different elements of the ESFM (states, transitions, guards, and test oracles) are generated and combined into the initial model in ModelJUnit notation. The completeness of the generated ModelJUnit-code varies for the different elements, and this is discussed in more detail in the following subsections. The implementation of the model generation technique described in this section is available as an open source project[**].

3.1. ModelJUnit Notation

In order to provide required background information for the model generation technique, this subsection presents the notation of the ModelJUnit tool for which the code is generated. The listing in Fig. 1 shows a model for a simple vending machine in ModelJUnit notation, adapted from [1]. The vending machine accepts 25 and 50 cent coins, and issues a product, through "vend," if 100 cents have been provided. After vending, the machine goes back to the initial state. The right-hand side of Fig. 1 shows the graphical representation of the code as provided by ModelJUnit.

ModelJUnit uses a specialized Java notation for describing the models. The `getState()` method is used by ModelJUnit to query the current state of the model. This information is used as feedback for the test generation algorithms. The `reset()` method is invoked when ModelJUnit starts the generation of a new test case. Typically, several test cases are generated from a model, with a given goal, such as satisfying a chosen coverage criterion. The `reset()` method must set the model into its initial state for the next test case (transition sequence) to be generated.

A second part of the model is described using Java annotations and naming conventions. This part defines the actual states, transitions and constraints, or transition guards, for invoking transitions. For example, Figure 1 shows five *states* for the vending machine, according to the values returned by

‡`http://czt.sourceforge.net/modeljunit`
§`http://www.junit.org`
¶`http://www.processmining.org`
‖`http://www.easymock.org`
**`http://noen.sourceforge.net`

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
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;}
}
```

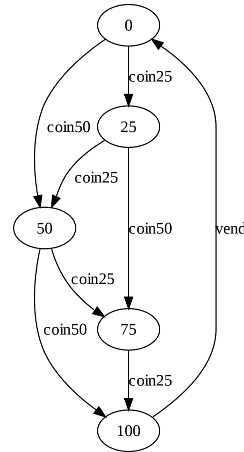


Figure 1. Example EFSM adapted from [1].

the `getState()` method, and updated by the *transition methods*, i.e., `coin25()`, and `coin50()`. Transitions are identified through the `@Action` annotation.

The *transition guards* determine when transitions can fire, defined in the methods `vendGuard()`, `coin25Guard()`, and `coin50Guard()` (Fig. 1). ModelJUnit identifies and associates these constraint functions with their corresponding transitions by matching each `@Action`-tagged transition method to a guard method with the same name but with `Guard` appended to the name. When this method returns true, the transition is permitted, and the related `@Action`-tagged method can be called. Otherwise, the transition is not permitted, and the related `@Action`-tagged method is not called.

In addition to the basic EFSM elements shown in Fig. 1, also two test-automation-specific elements are needed for the model. For generating tests, the transition methods must either record a test script (for offline testing), or directly provide input to the SUT (for online testing), such as performing a method call `sut.insert25()` in the `coin25()` transition method, which would execute the test case. We call this the *test harness*, as it binds the MBT tool to the SUT.

Test oracles are needed to validate the (expected) state in the model against the (actual) state of the implementation. For the `coin25()` transition method this can be provided as a simple assertion, i.e., `assert(money == sut.getInsertedCoins())`, inserted after the `money += 25` statement.

3.2. Case example used in this paper

For the rest of the paper, we use a running example based on a `Merger`-component of a maritime surveillance system. Here, we present the basic concepts of `Merger`, and in the following sections we use this to illustrate the different concepts related to our observation-based modeling approach.

`Merger` receives information broadcasts from ships called *AIS messages* [32] and processes them in order to form a situational picture of the coastal waters. The (simplified) architecture of this system

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

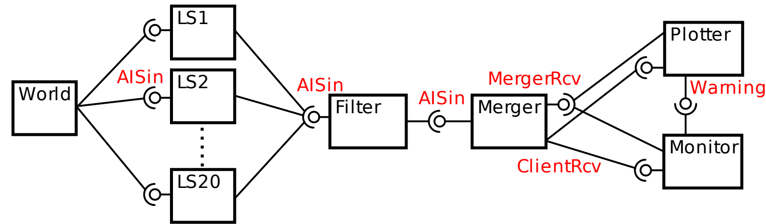


Figure 2. Architecture of the surveillance system used as example.

is displayed in Figure 2. The system comes with a specification in plain English defining behaviour and communication protocols of its components. The components are implemented in Java specifically crafted to be executed under Fractal [6], a component middleware platform.

The Merger acts as a temporary database of AIS messages, and client components can consult it for tracking information of a ship. It can also be configured by clients to be notified of certain ship events, and it is key to displaying ship tracks on the screen of the command and control centre.

3.3. 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. In order 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 [7, 10]. MBT is generally considered to be a black-box approach, based on the SUT's external interfaces and related specifications [33]. Similarly, our approach to generating model code is a black-box approach, and only observations from the external interfaces of the component are captured.

Because the model generated is only as complete as the execution scenarios used to produce the observations, it is important to verify that the scenarios include all the required behaviour of the SUT. Missing behaviour can be augmented with additional execution scenarios. We have used the visualizations provided by ProM and the invariant descriptions provided by Daikon as a basis for this analysis. These are discussed in more detail in [17].

To provide more powerful generation of the initial model for MBT, it is important to be able to classify the used execution scenarios and thus the captured observations by type. This is due to test oracles, which need to have meaningful classification of results as will be discussed in later sections. For example, typical test suites exercise both the nominal ("correct") behaviour of the SUT as well as its error handling functionality. For test oracles to make a useful assertion of correctness, they must be able to classify what is accepted and what is not. In order to achieve a useful classification in our `Merger` case study, we focused on using scenarios representing nominal behaviour as far as possible. Automated classification approaches such as [15, 20] could be applied to support this issue. However, it is out scope of this article. It is not a strict requirement to have such as classification, and even in the `Merger` case the scenarios also exercised some of the error handling properties, but a better

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

classification leads to more powerful generated (test oracle) model requiring less manual refinement as described in Sect. 4.

The information (observations) required to be captured in the trace includes the *messages passed* through the input- and output-interfaces of the SUT and the *SUT internal state*, when each message is passed. Messages are captured at the SUT's external interface. In order to identify different types of observations (related to SUT state, parameters, or return values) from the captured traces, for advanced processing and model generation, we add suitable identifiers into the trace. When middleware such as Fractal is used, this can be used to capture all component interactions, without having to instrument every component individually [16].

Accessing the internal state information typically requires testability support designed into the SUT, such as additional test interfaces, following [12], or serialization interfaces. For systems not supporting such a test interface, we can maintain an "artificial" state within the component that monitors the SUT external interfaces, by observing the inputs and outputs of the component and classifying them by type.

The `Merger` case study was conducted by running the complete system shown in Figure 2 with about 20000 real AIS messages. This produces the FSM in Figure 3. After consulting the (non-formal) specification, this was deemed as not a complete description of the SUT with respect to its expected functionality. To address this additional stimuli in the form of unit tests was used to complete the model with the missing functionality. The final FSM, including all these scenarios is shown in Fig. 4. This was considered a good and representative SUT behaviour with respect to the specification, and, thus, good enough a basis for generating the EFSM model.

3.4. Generating the basic model elements

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 SUT interface definitions are the variables used as input in this phase. Second, ProM is used to generate an FSM, and the FSM is analysed and processed to capture the interactions (states and transitions) for the EFSM. Third, Daikon is used to provide invariants over the SUT internal state and parameter data values, and the invariants are analysed to generate the relevant 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 in ModelJUnit notation. While the basic FSM and invariants are provided by existing tools, their further analysis, processing, integration and transformation into a complete EFSM suitable for MBT is done based on our own algorithms.

The basic model elements are `reset()`, and `getState()` methods, as well as the main method used to start the execution of the model with the MBT tool. These are illustrated in Listing 1, which shows examples of the most relevant parts of these generated methods. State variables for the model (the `List` objects in Listing 1) are generated for all variables identified to represent SUT internal state in the observations as described in subsection 3.3.

The generated reset method clears all model state and recreates the SUT objects to avoid side-effects between generated tests. The main model execution method (`modelJUnitTest()`) is generated in JUnit notation (`@Test`), in order to permit seamless integration with most IDE's (integrated development environments), that support reporting and analysis for JUnit tests. This model execution method is also generated to create the mock objects for the model (`mockClientRcv2`), and to store

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

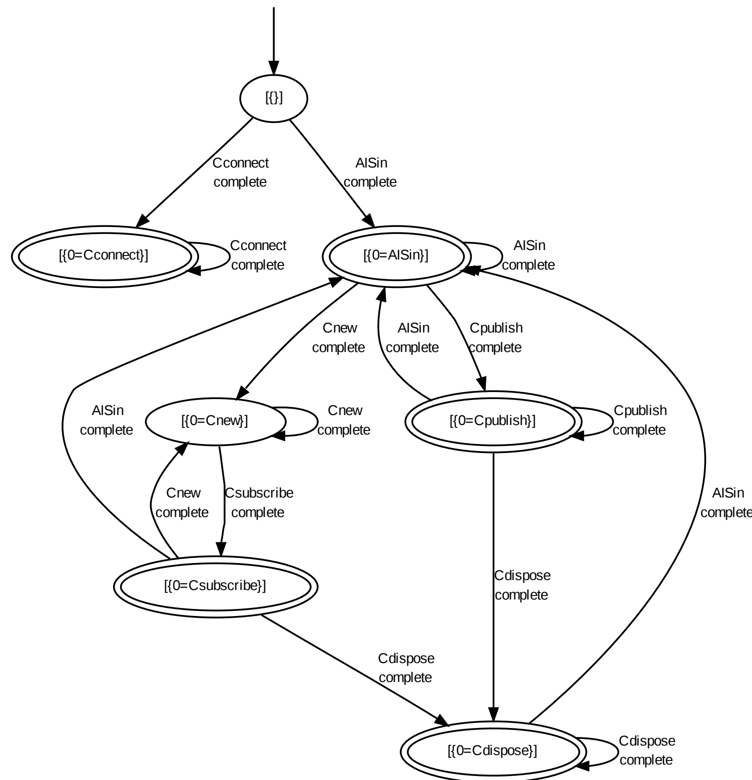


Figure 3. Merger FSM produced by ProM for the field data.

them in the model for the transition methods to access them. The generated mock objects are named according to the "mocked" output interface, i.e., `ClientRcv2`.

3.5. Transforming the FSM into code for MBT

The captured observations are transformed into an FSM with ProM's transition system miner component [3]. As described in section 3.3, the execution trace is based on input- and output-method invocations, made through the SUT's external interfaces. The FSM describes the SUT in terms of these method calls, where each message passed through one of the interfaces matches a state in the FSM. In order to provide powerful test generation based on an MBT tool, the states of the FSM cannot be used directly to describe the states in the EFSM. Instead, the differences between the input- and output-

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

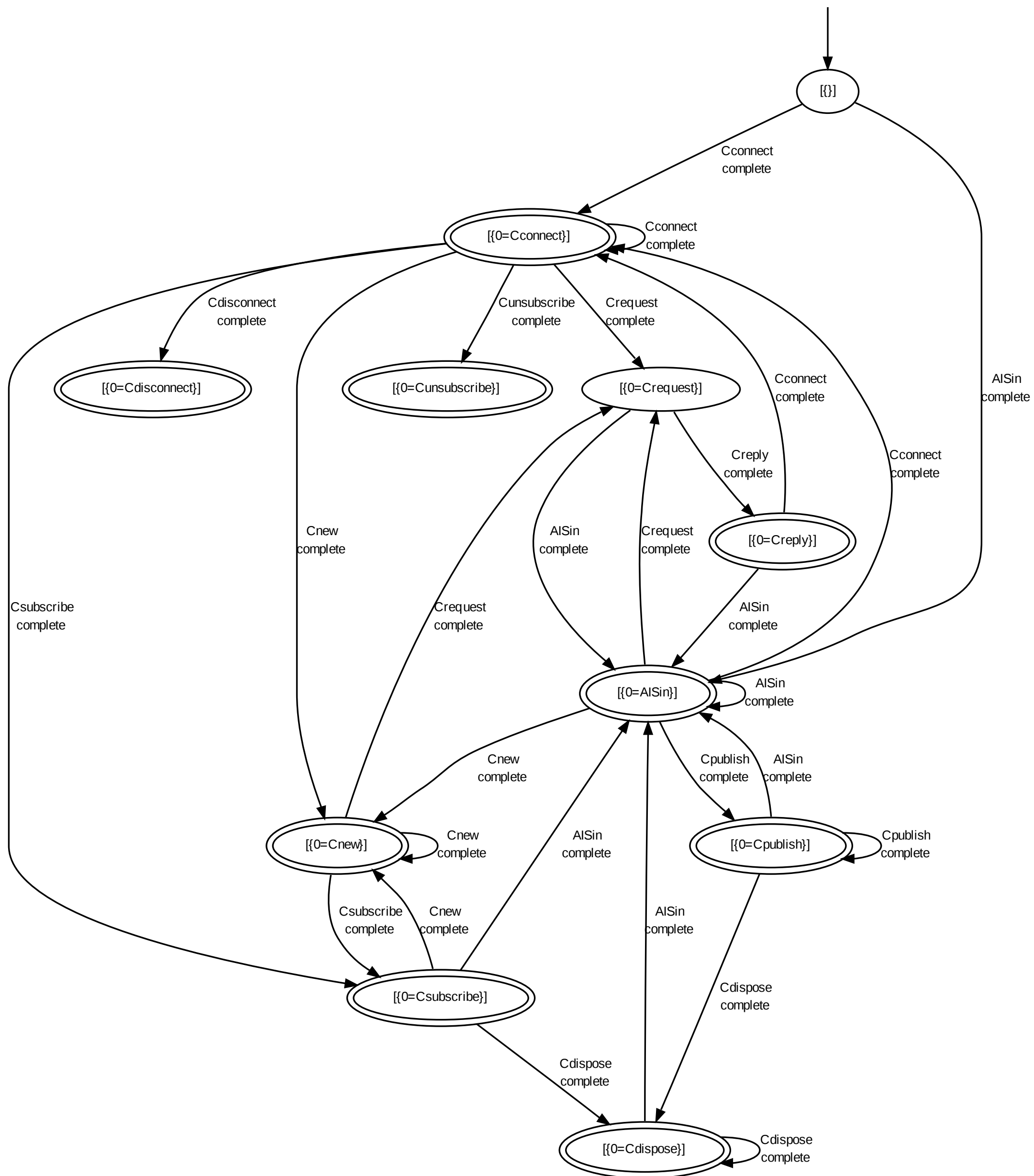


Figure 4. Merger FSM produced by ProM from combined field data and unit tests.

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
...
private int testIndex = 1;
private List Messages = new ArrayList();
private List Subscriptions = new ArrayList();
private List Clients = new ArrayList();
private AISMerger aISMerger;
private ClientRcv2 mockClientRcv2;
...
@Test public void modelJUnitTest() throws Exception {
  mockClientRcv2 = createMock(ClientRcv2.class);
  Tester tester = new RandomTester(this);
  ...
  tester.generate(2000);
  ...
}
public void reset(boolean b) {
  state = "";
  System.out.println("- TEST "+testIndex+" -");
  testIndex++;
  Messages.clear();
  Subscriptions.clear();
  Clients.clear();
  EasyMock.reset(mockClientRcv2);
  try {
    aISMerger = createAISMerger(mockClientRcv2);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
public Object getState() {
  return state;
}
...
```

Listing 1. Generated reset, getState, and main execution methods for Merger.

messages in the FSM need to be considered. This is a good example for the modeling expertise required by a user of this method for producing effective models for MBT.

In order to produce a model more suitable for MBT, we consider a state transition to be triggered by an invocation of an input-method to the SUT. For each input-method in the FSM, a matching `@Action`-method is generated in the model code. Listing 2 shows two example `@Action` transition and transition guard methods for the `Merger` component.

The basic elements generated for each `@Action` transition method are also shown in Listing 2 (for `Crequest`). The state of the model is always set to a name matching the taken transition (`this.state = "Crequest"`). This allows the tool to use its model coverage algorithms to cover different combinations of the interaction sequences. The name of the state transition taken is printed out (`System.out.println("CREQUEST")`) in order to make it easier to follow the paths that the MBT tool takes while it generates tests from the model, e.g., for debugging.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

```
@Action public void Crequest() throws Exception {
  this.state = "Crequest";
  System.out.println("CREQUEST");
  replay(mockClientRcv2);
  ReturnStatus rv4 = aISMerger.Crequest(Crequest_p0(),Crequest_p1(),Crequest_p2());
  assertEquals("ok", rv4);
  verify(mockClientRcv2);
  EasyMock.reset(mockClientRcv2);
}
public boolean CrequestGuard() {
  if(Clients.isEmpty()) return false;
  return true;
}
@Action public void Crequest_Creply() throws Exception {
  this.state = "Crequest->Creply";
  System.out.println("CREQUEST->CREPLY");
  expect(mockClientRcv2.Creply((AISMessage)anyObject()))
                                .andReturn("ok");
  replay(mockClientRcv2);
  ReturnStatus rv5 = aISMerger.Crequest(Crequest_p0(),Crequest_p1(),Crequest_p2());
  assertEquals("ok", rv5);
  verify(mockClientRcv2);
  EasyMock.reset(mockClientRcv2);
}
public boolean Crequest_CreplyGuard() {
  if(Clients.isEmpty()) return false;
  if(Messages.isEmpty()) return false;
  return true;
}
...
private String Crequest_p0() {
  return (String) randomItemFrom(Clients);
}
private int Crequest_p1() {
  return (int)1.0;
}
private byte Crequest_p2() {
  return (byte)1.0;
}
...
```

Listing 2. Generated reset sample transition (@Action), guard and parameter value generation methods for Merger.

The next step is the generation of the expected interactions within a transition. They are based on the FSM and the categorization of each message in the component interface into an input or output message. This classification is based on the input- and output-interface definitions (Java classes) of the SUT. The FSM is analyzed according to this information, and each input-state (message) is associated with outgoing transitions to any output-state (message).

A number of `@Action` transition methods is generated for each input message, one for the input message alone, and one for each possible output message to which it has an outgoing transition. For example, the FSM shown in Figure 4 has a state `Crequest`, which can either go to `Creply` or to

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

AISin. As only Creply has been classified as an output message, we obtain in Listing 2 two @Action methods: Crequest() corresponding to the input message itself when the given input produces no output message (no Creply), and Crequest_Creply(), corresponding to the input message followed by a Creply output message.

For each @Action transition method, an *interaction oracle* is also generated. These are use the interfaces provided by the generated EasyMock objects to verify that the expected transitions do happen, and only happen as expected. In the ones that include an expected output transition from the input transition, an expectation is set that this output transition actually happens. For the ones without expected output transitions, an expectation is set that no output transition occurs in the SUT.

In the Crequest_Creply() method the expectations for the output method interaction are set as expect(mockClientRcv2.Creply((AISMessage)anyObject())).andReturn("ok");. This means that we expect the SUT (Merger) object to call the Creply() method of mockClientRcv2 with a parameter of type AISMessage, and when this happens the mock object should return the value "ok" to the SUT. Once the expectations are set, a call is made to the input method of the SUT that corresponds to the state transition method being executed. In the case of Crequest_Creply() it is the Crequest() method. Generation of the return values and the parameter value template methods will be discussed in more detail in the subsection 3.6.

Finally the results are verified, i.e., the interaction test oracles are invoked, in the form verify(mockClientRcv2), using the name of the corresponding mock object. This checks, that all expectations set for the mock object are met, and no additional interactions are performed.

3.6. Transforming the invariants into code for MBT

The second model used in the generation of the EFSM code is the set of invariants provided by Daikon, describing the properties of the parameters and return values of the input- and output-interface method invocations for the SUT, plus their relations to the internal state values of the SUT. They are used to generate possible return values for the mocked output message sequences, parameter values for input messages, and guard conditions for transitions.

Daikon can output the invariant information in many different formats for testing [11]. However, none of these formats is directly usable for our purpose. Therefore, we use the basic textual output, parse and further process it, and finally generate code out of it. We also use our own customized Daikon trace format, permitting more advanced analysis of the invariants for MBT. Similarly, for guard conditions, we provide our own generalization of the Daikon invariants for more powerful guard generation. An example of the basic Daikon invariant output used as input for our algorithms is shown in Listing 3 for Crequest. This illustrates how the customized trace format discussed in section 3.3 can identify invariants related to the different model elements (state, return values, parameter values). Here they are identified by the postfix appended to the name in the trace as visible in listing 3. Additionally, return value program points are identified by postfix _EXIT in order to work around some limitations of the basic Daikon format.

3.6.1. Transition guards

For generating the transition guards, each internal state-related invariant is taken for processing. For example, in Listing 2, the guard for the Crequest_Creply() transition method is the Crequest_

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

```
====================================================
Crequest:::ENTER
1.shipID?1 == AISType?2
2.shipID?1 == size(Clients?g[])
3.size(Clients?g[])-1 == size(Subscriptions?g[])
4.clientName?0 == "myclient"
5.shipID?1 == 1
6.Clients?g[] == [myclient]
7.Clients?g[] elements == "myclient"
8.Subscriptions?g[] == []
9.size(Clients?g[]) == 1
10.clientName?0 in Clients?g[]
====================================================
Crequest_EXIT:::ENTER
ReturnStatus?r == "ok"
====================================================
```

Listing 3. Sample Daikon output for Crequest.

`CreplyGuard()` method. Two guard checks have been generated for this transition, each defining a condition that needs to be met for this transition to fire.

We started by turning all the related invariants for a transition into guard checks as expressed by the Daikon output. Only the values related to the internal state of the SUT are available when guard conditions are evaluated, so any invariants related to parameter values can be discarded as non-relevant. In Listing 3, this leaves us with the invariants on lines 3 and 6-9.

Turning all these into guard statements leads to five transition guards, i.e., for invariants 6 and 7, guard statements must be generated for checking that the state variables always contain only elements of type `myclient`, and only contain one of these. Similarly, for invariant 9, a guard must be generated to check that the Client's state variable always includes exactly one item.

Turning the Daikon invariants into transition guards, results in a number of useless guard statements, and the useful ones become overly constraining. For instance, analysis of invariants 3 and 6-8 reveals that they represent random properties of the used execution scenarios, and invariant 3 is a combination of invariants 8 and 9. Associating the two state variables is not meaningful as they are not really related.

Another example is invariant 8, stating that the list of subscriptions should always be empty at this point. Although true for the used execution scenarios, it is not a correct requirement as subscriptions are allowed when requests are made. The case where subscriptions exist while making a request is simply not contained in the execution scenarios used.

Only invariant 9 is useful, although, overly constraining, in stating "there should be a connection, but no more than one," whereas the correct invariant, according to the specification, should state "at least one connection." These examples illustrate that too many and too constraining conditions are generated from Daikon output when used directly.

Instead, we use abstractions over the Daikon invariants to provide more powerful transition guard generation. After doing a complete model refinement for testing and verification (described in Sect. 4) for the `Merger` component, we found that all useful guard statements were related to the size of single model state variables (such as invariants 8 and 9). The correct check turns out to be "the state variable has some content or not."

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

To provide more powerful guard generation, we devised a more specific algorithm that only generates checks for invariants related to the size of the state variables. Daikon produces several different types of invariants describing the contents of array variables, including size and content. If it can be inferred from these invariants that the size of that state variable is always greater than zero, a guard is generated to check that this variable has some content. This produces much better results compared to direct use of Daikon invariants. The results and their validity will be discussed in more detail in Sect. 5.

3.6.2. Object values and creation

Test automation requires the creation of SUT domain objects to provide test input and expected output. We generate templates for providing these objects in the model. The templates include the feasible values inferred from the invariants provided by Daikon for these objects. Daikon in- and output is limited to strings, and, thus, best suited for describing primitive objects only.

Generation of non-primitive objects cannot be fully automated since it is impossible to determine how primitive values in the Daikon invariants have to be mapped to previously unknown domain objects and their constructors. Instead, the value in the invariant model is provided to the user as a basis for manual refinement. Listing 2 shows this as `"ok"` in both transition methods `Crequest()` and `Crequest_Creply()`. The same applies to the return value given to the SUT when it invokes the `mockClientRcv2` mock object in `Crequest_Creply()`, which is shown as `.andReturn("ok")` in Listing 2. During refinement, the `"ok"` must be amended to create an actual domain object matching the invariant value, as will be shown in section 4. This illustrates the domain knowledge required of the user.

For the parameter value generation, the relations of parameter values to the internal state of the model are also considered in addition to providing invariant values as a basis for object creation. This is illustrated in Listing 2 by `Crequest_p0()`, and `Crequest_p1()`. For `Crequest_p1()`, invariant 5 in Listing 3 describes this value as a constant of 1 and `Crequest_p1()` is generated to provide this value. For `Crequest_p0()`, invariant 10 in Listing 3 describes this value as always being one from the list of connected clients (from the `Clients` state variable). To provide suitable values, `Crequest_p0()` is generated to pick an item from this state variable as a parameter.

4. TESTING AND VERIFICATION METHOD

The process of using a *generated* model for testing and verification is different from the traditional model-based testing process. We call it a verification process as it allows verifying the completeness and correctness of the (implementation) model and the specification in relation to each other, and a testing process as it generates new test cases to exercise and evaluate the SUT behaviour.

Traditionally in MBT, the user takes the SUT specification as a basis to create a test model for the system. In our approach, the initial model is generated based on the captured observations, leading to an advanced starting point for the partially manual process of converting it to the final model to be used for testing and verification with a MBT tool. Checking whether the model contains all expected behaviour as specified, and whether this behaviour is correctly implemented, is an iterative process. Basic requirement for this process is the SUT specification for verifying and refining the model.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

The method of using the initial generated model is outlined in the following:

1. Disable all state transitions (set guards to false).
2. Choose an initial state that can be reached from SUT startup.
3. Evaluate this generated state transition and related guard statements for correctness w.r.t. the specification, making any modifications necessary.
4. Enable this state transition (correct the guard).
5. Execute the model with the MBT tool.
6. Analyse the root causes of any errors reported, and fix either the SUT, specification or the model accordingly.
7. Choose the next state that can be reached once the initial state transition is enabled and verified for correctness, and fix any errors found.
8. Continue this process from step 3 until all state transitions have been processed.
9. If any generated state transition remains disabled, or is not described in the specification, check why the transition exists in the implementation. Amend the specification if it is a correct transitions, amend the implementation if it is incorrect.
10. Finally, evaluate the complete model with all the generated state transitions containing all expected (input-output) transitions included in the specification. If any are missing, check if this is due to restrictions of the used execution scenarios or due to missing implementation. Fix the cause.

According to our experience gained, it is best to start by focusing on one state transition at a time. This is achieved by disabling all other transition guards (step 1). Based on the SUT specification, an initial state can be selected for analysis (step 2). This state must be reachable from the initial starting state. When a state is chosen for analysis, its transition guard is enabled (applying correct constraints) to allow the MBT tool to explore it for test case generation.

Evaluating a state consists of checking that the state has the correct transition guards, so the transitions can fire as expected , and that the transition is complete with regards to all required elemenents(step 3). In other words, evaluating that the internal state of the model is correct and allows executing the transition as expected, making it possible, for example, to provide correct parameter values (such as picking a value from a state variable as shown by `Crequest_p0()` in listing 2). To enable analysis of the chosen transition with the help of the MBT tool, its guard statement must first he enabled (step 4). Evaluating a state also consists of checking the transition method itself and checking correct values are provided, adding domain object creation, as well as updates to the model internal state as a result of any successful transition. The extent of these modifications is discussed in more detail in section 5.1. One of the advantages of this method and the given model is that it is possible to execute the model with the MBT tool at any time during this process to verify the expectations and changes made to the model (step 5). This will reveal any inconsistencies between the refined model and the implementation, providing precise information to help with the analysis of the cause of any possible failures (step 6).

Once a state transition is considered to be complete and correct, more state transitions can be enabled, one at the time in an iterative manner (steps 7-8). Enabling one transition allows enabling another following transition as the SUT state is updated by the previous transitions. Repeating this process for all states, eventually produces the complete model. Additional errors can be discovered

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

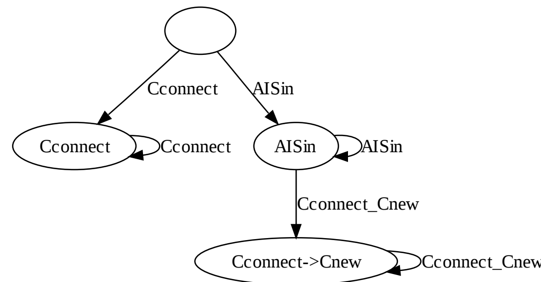


Figure 5. Merger EFSM with 3 states enabled and as visualized by ModelJUnit.

as more states are added and more complex interactions are produced by the MBT tool, potentially requiring more analysis of previous states in the model.

Finally, the complete model must be checked against the SUT specification to assess if there are any excess transitions left over that should not be (step 9) or if any required transitions are missing (step 10). That is, verifying the completeness and correctness of the implementation and the specification.

In order to illustrate the result of refining an initial generated model to form a completely verified and tested model, Listing 4 shows a fully refined version of the code from Listing 2. This was produced as an end result of the `Merger` case study. In the following, we describe the modifications made during the process of applying the described method on this part of the generated `Merger` model.

Based on the FSM provided by ProM, we can only make an assumption that a single output message is expected for a single input message. However, it is possible to receive several output messages back for a single input message. For example, the transition method `Crequest_Creply()` in Listing 2 should produce one or more replies depending on its internal state. However, based on the FSM, the interaction oracle (mock object) was generated to only expect one reply. In Listing 4, this has been amended through adding `.anyTimes()` to the end of the expectation for the `Crequest_Creply()` transition. This refinement is implicitly "required" by the MBT tool by showing where expectations have failed.

Examples of simpler refinements for non-primitive objects are the changes from `"ok"` to `ReturnStatus.ok`. An example of refining the generation of a primitive value is shown in Listing 4, where the `Crequest_p1()` method must return an `id` value which was received previously by the SUT. It must be refined manually to take one of the messages from the model state list variable `Messages` and return the `id` values of this message.

The executed model can also be visualized at any time with ModelJUnit. This is convenient in order to get a visual representation of the generated model thus far. Figure 5 displays such visualization of the `Merger` model in which the first three states are enabled.

This visualization can be used to evaluate the final generated model w.r.t. completeness according to the specfcation. Each oval (state) shows the input-output transitions of the model, where expected output is shown to be followed by the input. If no output is expected in a state, this is shown as a single input element. The arrows indicate the sequential order in which these transitions can occur. Since

Copyright © 2007 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
...
@Action public void Crequest_Creply() throws Exception {
  this.state = "Crequest->Creply";
  expect(mockClientRcv2.Creply((AISMessage)anyObject()))
                       .andReturn(ReturnStatus.ok).anyTimes();
  replay(mockClientRcv2);
  ReturnStatus rv5 = aISMerger.Crequest(Crequest_p0(),
                                        Crequest_p1(),
                                        Crequest_p2());
  assertEquals(ReturnStatus.ok, rv5);
  verify(mockClientRcv2);
  EasyMock.reset(mockClientRcv2);
}

public boolean Crequest_CreplyGuard() {
  if(Clients.size() < 1) return false;
  if(Messages.size() < 1) return false;
  return true;
}
...
long msgTime = 0;
int nextMsgId = 1;
private AISMessage AISin_p0() {
  AISMessage message = new AISMessage((byte) 1, 0,
                             nextMsgId, new Date(msgTime));
  nextMsgId++;
  msgTime += 1000;
  Messages.add(message);
  return message;
}
...
private String Crequest_p0() {
  return (String) randomItemFrom(Clients);
}

private int Crequest_p1() {
  AISMessage msg = (AISMessage) randomItemFrom(Messages);
  return msg.getUserID();
}

private byte Crequest_p2() {
  return (byte)1.0;
}
...
private String Cdisconnect_p0() {
  String client = (String) randomItemFrom(Clients);
  Clients.remove(client);
  Subscriptions.remove(client);
  return client;
}
...
```

Listing 4. Refined versions of methods in listing 2.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

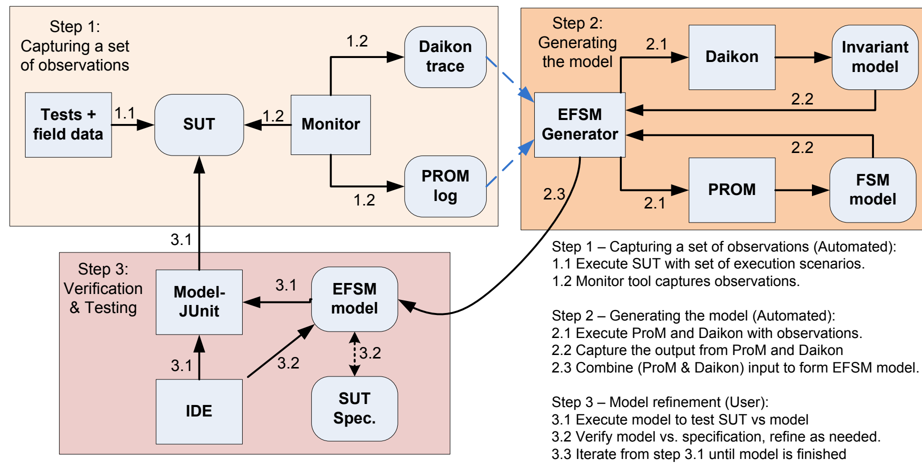


Figure 6. Overview of the complete proposed approach.

there are potentially many (input-output) state transition sequences, this is most useful for identifying the included input-output transitions as well as the initial start states. The visualization shown is a result of our chosen state representation in the generated model (the `state` variable contents).

Finally, figure 5 presents a complete overview of the whole approach proposed in this paper, including capturing the observations, generating the initial model, and using this as a basis for SW testing and verification.

5. CASE STUDY EVALUATION

In this section, we discuss the experiences and lessons learned from applying our approach and method to the example system described in section 3.2. The aim of our case study evaluation was to evaluate the usefulness and effectiveness of the presented approach. We start by discussing the precision and recall for our model generation technique, that is, how much of the different parts of the model are provided by the code generator. We also discuss the usefulness of applying our approach to testing and verification of the `Merger` component.

In order to evaluate the outcome further, we briefly discuss the application of our approach to a second component of the same system, called `Filter` (see Fig. 2). The state representation is similar to `Merger`. It maintains a list of processed AIS messages and filters out duplicate entries from overlapping surveillance areas. This component was studied after completing the `Merger` example, in order to evaluate whether the techniques can be generalized, in particular, our optimization of the transition guard generation with invariants.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table II. Number of different elements in the final (complete) model.

Model Element	Merger	Filter
Transitions	13	2
Interaction oracles	13 (4)	1
Mock objects	1	1
State variables	3	1
Reset methods	1	1
Main execution methods	1	1
Parameter values	14	2
Input methods	6	1
Output methods	4	1
Return value oracles	7	0

5.1. Precision and Recall

Table II lists the number of elements present in the complete `Merger` and `Filter` models. *Transitions* are input-output transitions represented by the `@Action` tagged methods in the model. `Merger` has one output interface with four *output methods*. This interface is represented in the model by a *mock object*, which is used to verify the correctness of interaction sequences (as *interaction oracles*). Moreover, `Merger` has two input interfaces with six *input methods* in total. 14 *Parameter values* are required for the input messages generated for the SUT. 7 *Return value oracles* assert the values received as return values from the input methods in the transitions.

The precision and recall for model generation in the `Merger` case are shown in table III. *Full* means the generated code does not need to be changed, and *Partial* means that an initial version of the required code is generated and has to be amended, but the generated code hints to what it should be changed to. For example, when a transition guard checks that a state variable is always of size 1 or 3, it is easy to see that it should, in fact, check it has some content (size >= 1). “Missing” means something is not generated, although, it should. “Redundant” means that some code is generated, but must be deleted, because it is not relevant to the model. In the following subsections, the values obtained are discussed `Merger` case. Finally, we summarize the results for the `Filter` case.

The most interesting entries in the table are the *state transition methods*, *transition guards*, *parameter values*, *state updates*, and *test oracles*. The remaining entries are mostly static items always completely generated as described in section 3.4 and illustrated in Listing 1.

5.1.1. State transition methods

State transition methods are generated completely for all input represented in the traces. However, we discovered that `Merger` had an error in its implementation, where the given input did not lead to the expected output, due to missing functionality. Hence, one required state was not generated as it was not implemented in the SUT, and no observation captured from the SUT execution could therefore

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table III. Merger precision and recall breakdown by model elements.

Model Element	Full	Partial	Missing	Redundant
Reset method	100% (1/1)	0%	0%	0%
State variable definitions	100% (3/3)	0%	0%	0%
State variable updates	0%	0%	100% (6/6)	0%
Main execution method	100% (1/1)	0%	0%	0%
State transition methods	91% (10/11)	0%	9% (1/11)	15,4% (2/13)
Transition guard checks (Daikon)	0%	76,9% (10/13)	23,1% (3/13)	68,7% (22/32)
Transition guard checks (Custom)	76,9% (10/13)	0%	23,1% (3/13)	9,1% (1/11)
Interaction oracles (no output)	100% (9/9)	0%	0%	0%
Interaction oracles (output)	50% (2/4)	50% (2/4)	0%	0%
Mock object return values	100% (4/4)	0%	0%	0%
Parameter values	21,4% (3/14)	0%	78,6% (11/14)	0%
Return value oracles	71,4% (5/7)	28,6% (2/7)	0%	0%

describe this state transition. This highlights the importance of analyzing the provided model properly w.r.t. the specification for the required transitions, as some may not be implemented as they should.

Two “Redundant” state transitions were generated. In the first instance, the information for a ship is discarded if no more messages regarding this ship are received for some period of time. This is not triggered by any given input message, but by a timeout, and it can be observed in the trace at any arbitrary point in time. This was a part of the large execution scenario with field data as input. This also highlights how some specific functionality needs to be tested for with different types of approaches.

In the second instance, the reason for generating a “Redundant” state transition is not so obvious. We could trace it to execution scenario instantiating the complete system that processes the field data and spawning several independent threads on the go, some of which cause an interleaving of the captured observations, producing observations for sequences not relevant for the actual SUT behaviour.

5.1.2. Transition guards

We separate two types of transition guard checks, discussed in Sect. 3.6, i.e., *Transition guard check (Daikon)* and *Transition guard check (Custom)* in Table III. The first one refers to the original Daikon invariants, whereas the second one refers to our optimized method that only considers the invariants related to the size of the state variables. Since both use the same set of invariants as a basis, the numbers of “Partial” for the first entry (“Daikon”) and “Full” for the second entry (“Custom”) are the same.

First, Daikon generated a large number of redundant invariants not relevant in any way to what is required for our model. Seven out of ten partially or fully provided transition guards are duplicates generated by Daikon. After removing the most obvious ones, the fraction of redundant invariants was still 68,7% of all the invariants provided by the “pure Daikon” approach. It requires a lot of user interaction in order to devise the “correct” invariants, and it is, by no means, satisfying.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Applying the "Custom Transition guard checks" proved to be much more convincing, with the number of redundant guard checks of 9.1% of the overall guards provided by Daikon. It must be noted, however, that this technique can still produce redundant checks when invariants for irrelevant state variables are generated as a by-product of the observations made from the execution scenarios. However, in this case the number of redundant guards was acceptable leading to much less analysis and correction work to be done by the user of this approach.

For the "Missing" transition guard checks, different reasons could be identified. One condition involved checking an "OR-relation", a composed expression, not (yet) supported by Daikon. Instead, Daikon simply reports that neither of the invariants in the composed expression hold. Another condition not generated could be attributed to (likely) faulty behaviour of Daikon. The invariant was always apparent in the captured observations, but never reported by Daikon. It is not clear why the last transition guard is missing in Daikon, as values related to this transition are numerous in the trace, and performing detailed analysis on this was out of the scope of our study.

5.1.3. Parameter values

Parameter values are difficult to infer, due to the complexity of non-primitive objects, and due to limited execution scenarios that do not typically exhibit the full range of input values permitted. The successfully generated values were based on either primitive values, or on relations for which the parameter value had to be a match to one that was already stored in the SUT state variables. Parameter generation is still an issue to be addressed, for example through including domain objects serialized from previous SUT executions as described in [27], or mining object specifications from SUT source code and executions, as proposed by [30]. However, this open issue must be addressed in future work.

5.1.4. State updates

In addition to domain object creation, also related model state updates need to be added by the user. For example, when a new message is provided to `Merger`, the processed Message object must be added to the provided `Messages` model state variable. This is illustrated in Listing 4 in the The `AISin_p0()` parameter value creation method, that adds the new processed message to the state variables `Messages`. This also illustrates the need for the user to add the creation of domain objects manually. In this case, each of the messages needs a unique id value and a unique timestamp.

These model parts related to state updates are not automatically generated due to limitations of automatically knowing suitable representations needed for domain objects, and due to the limitations of the underlying models used to generate the code. Daikon is intended to provide invariants over program points, whereas this requires support for combining data over several points, such as the provided parameter values, and their relation to the SUT internal states before and after messages are processed. To address this issue, invariant detection could be extended to automatically cover this type of more complex inter-relations. However, we did not find means to easily extend Daikon to do this and leave this out of the scope of this paper.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table IV. SUT coverage breakdown by execution scenarios and tests for Merger(M) and Filter(F).

Source	Statements	Methods	Conditionals	Paths
M:Unit tests	53,5% (76/142)	64,5% (20/31)	38,7% (24/62)	6
M:Data	61,3% (87/142)	64,5% (20/31)	51,6% (32/62)	27
M:Data+Unit tests	77,5% (110/142)	87,1% (27/31)	61,3% (38/62)	33
M:EFSM	64,1% (91/142)	67,7% (21/31)	48,4% (30/62)	87
M:EFSM+Unit tests	65,5% (93/142)	67,7% (21/31)	51,6% (32/62)	92
F:Unit tests	38,2% (21/55)	23,5% (4/17)	32,1% (9/28)	2
F:Data	52,7% (29/55)	35,3% (6/17)	50,0% (14/28)	17
F:Data+Unit tests	52,7% (29/55)	35,3% (6/17)	50,0% (14/28)	18
F:EFSM	45,5% (25/55)	29,4% (5/17)	39,3% (11/28)	79
F:EFSM+Unit tests	45,5% (25/55)	29,4% (5/17)	39,3% (11/28)	79

5.1.5. Test oracles

As described in earlier sections, two different types of test oracles were generated, interaction oracles and return value oracles. There are two types of interaction oracles listed in table III, ones marked "no output" and ones marked "output". Here "no output" means the provided input is not expected to provide any output. For "output", an input-output interaction sequence is expected. All interaction oracles were generated with the correct expectations. However, half of these (listed as "partial" for the "output" cases in table III) were missing the expectation that the input message can be followed by any number of output messages. Thus it required the addition of `.anyTimes()` to these expectations.

For return value oracles, as well as mock object (interaction oracle) parameter values, the full generation means that the provided model gave only one option and it was the correct one. For the partial ones, the provided options included the correct one but also additional ones. The option here refers to the string identifier used for the domain object in the observation trace. For reasons discussed before, full domain objects are not generated.

5.2. Test coverage

Although we applied our model generation approach completely as a black-box approach, we also had access to the source code and were able to gather code coverage metrics for the different test scenarios. These are shown in Table IV. Here unit tests refer to the six unit tests used as execution scenarios. Data refers to the set of field data and the system application that was used to process the data through the complete system. Model 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, in other words, it describes the combinations of input-output sequences taken during SUT execution.

In Table IV it is visible that the tests generated from the EFSM model provide a significant increase in coverage over the used unit tests. *Data* has a high coverage value but it can not be considered a test as it simply executes the SUT without any assertions. As it has no test oracle it is not actually testing

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

anything. However, it is still a useful exploration of the SUT behaviour for capturing observations. The complete coverage of the execution scenarios used to capture the observations is shown here as *Data+Unit tests*. For the difference between the code coverage of this set and that of the *EFSM* set, a detailed analysis revealed that the additional code covered by *Data+Unit* tests is due to the larger amount of setup involved in putting the complete system together vs. the minimal setup with mock objects in the *EFSM* set.

The total coverage of all tests is shown by *EFSM+Unit tests*. This shows that although *EFSM* does provide significant increase in coverage alone, the *Unit tests* still provide an additional increase and thus the tests in the *EFSM* set and *Unit tests* set are complimentary, as is to be expected for black-box testing and unit testing. This also provides validation for the use of unit tests as execution scenarios for model generation in MBT.

Table IV shows the *Unit tests* and the *EFSM* paths to be mostly different. This is most likely due to the MBT tool generating longer sequences, while the unit tests only produce short sequences. Therefor the paths in the unit tests are likely covered also by the *EFSM* set but as the number of paths here is based on complete unique paths, it does not count two paths as one if one is embedded inside another.

For the actual behaviour related code, the *EFSM* set outperformed the others by a small percentage due to the generalization of the generated model in the verification and testing refinement phase. This generalization led to execution of additional parts of the code.

The biggest difference is in the paths metric, *EFSM* outperforms the others by a factor of more than two. This is what is to be expected from a MBT tool that is intended to generate complex interactions to test the SUT, and could be even further increased through MBT tool parameters.

5.3. Mutation testing

In addition to code coverage analysis, we also conducted another coverage measured in the form of mutation testing of both the `Merger` and the `Filter` components. The mutation testing was focused on finding how many of the generated mutants are "killed" (discovered) by the used test cases. A mutant is a semantically different modification in the Java class of the component. In our case these mutants were generated with the μJava[††] tool. Some of the generated mutants are “equivalent”, meaning that their behaviour is exactly the same as the original version (e.g.: increment a variable never used after, in/decrement a huge arbitrary constant, modify how a hashcode is computed). We manually categorized the generated mutants to ones that really modify the behaviour of the SUT and to ones that do not.

To evaluate the effectiveness of the generated and refined model in killing these mutants, the completely refined model as provided from the previous sections was used to generate tests and these tests were executed to kill the mutants. Each provided mutation was applied separately and the tests were executed to see if the changes are detected (if the mutant is killed). The results are shown in Table V. It should be noted that this does not simulate the case of starting from initial model generation for each mutant but simply evaluates the power of the complete and final model in killing mutants, which places this mutation experiment closer to the domain of regression testing.

[††] `http://cs.gmu.edu/~offutt/mujava/`

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table V. Mutation test results.

Source	True positive	False positive	True negative	False negative	Total
Merger unit tests	51	16	50	0	117
Merger EFSM	51	15	51	0	117
Filter unit tests	36	7	21	0	64
Filter EFSM	36	5	23	0	64

When a test finds no errors (the SUT is considered to operate fine), the result is termed "positive". When an error is reported, the result is termed "negative". "False positives" are the mutations which are said to be working 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.

Table V shows that the final model provides minimal gain over the initial unit tests. 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 model outperforms the unit tests in correct categorization of mutants with actual modified behaviour only by a slight margin.

Although the differences in code and mutant coverage do not seem to be high, the different paths generated with the MBT tool combined with the manual verification and testing process proved very useful in this case, finding several previously uncaught bugs in the implementation as will be described next.

5.4. Errors Discovered in the Merger

In the process of refining the complete model for `Merger`, we found several errors in the SUT implementation. These errors can be classified to different types including mismatches between implementation and specification, ambiguities in the specification, and problems in the design that cause errors under certain conditions.

Problems in the SUT design were related to assumptions it made about its environment. The design made the assumption that one client component would never have several connections at the same time with the `Merger` component. This functionality was not covered by the initial execution scenarios and the SUT specification was also not detailed to the level to define this type of details. Instead, this was revealed by the complex input and interaction sequences generated by the MBT based on the refined model. This also lead to a requirement of refining the specification.

Mismatches identified between implementation and model were wrong return values received from the SUT, discovered by the return value oracles, and incorrect or missing transitions in the SUT, discovered by the interaction oracles or by inspection of the FSM vs. the specification. In the case of return values, making a connection would always return “ok”, regardless of the parameter provided, and whether a connection was successful or not. This was a clear violation of the specification, which states that `Merger` should return error codes. Although this functionality was exercised in the initial execution scenarios, they did not use sequences and assertions that would reveal this error. The various interaction sequences generated from the model, along with the test oracles did reveal it.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Another issue detected was a missing specification item, about queries on ships that do not exist. The generated model issued an "ok", but the implementation returned an error code. The specification made no statement how this should be handled by `Merger`. In this case, this highlighted a need to update the specification and then re-evaluate the model vs. the implementation. This shows how having two different "implementations" of the specifications (the SUT and the model), makes it more likely to reveal the ambiguities and misunderstandings in the specification.

Protocol issues of the `Merger` were discovered through the interaction oracles and through inspection of the FSM. The specification states that "if a client is subscribed to a ship for which data exists, the Merger should immediately publish this data to the client," which it did not. This problem was found when comparing the model and specification and could be verified by inspecting the FSM's shown in Figure 4. This also highlights the importance of carefully comparing the generated model to the specification. As discussed before, a generated model only contains the information observed in the execution scenarios, excluding any unimplemented features. These can only be revealed by verifying the model against the specification to see that it describes all required functionality and is correct.

The subscription code contained another problem that was discovered by the interaction oracles. Subscribing to a ship for which no messages had been received so far, caused the loss of data through a missing output message. This was an error in the implementation. Again, the required interactions to create the initial FSM to test this were present in the used execution scenarios, but this was only discovered by the thorough checking done in all states by the MBT when generating tests based on the final model.

These issues could be resolved, eventually, by amending the `Merger` code following the specification and the refined model. 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.

5.5. Filter

As already shown in Table II, the `Filter` case study was much simpler than the `Merger` case study. This is also reflected in Table VI, showing the precision and recall values for the `Filter` case study. Since there are only one or two of each model element needed and generated, the values are mostly 100%. This is also visible in the number of invariants generated, where only one invariant is generated and this is turned into the correct guard condition by our customized invariant processing algorithm. This lack of invariants is also likely due to the small number of variables present in the observations. As the messages observed do not have any return values, there are no mock object return values or return value oracles needed or generated in this case. This is a simple example, but serves to provide evidence that the approach is usable over more than just one (`Merger`) component, and also on simple systems.

5.6. Limitations of the approach and the case study

We realize that the results can not be generalized as such to all possible systems, as their representation of state can be different (such as primitive values) or different types of invariants may be important. To provide a more comprehensive analysis and support for different types of state representations, experiments with different types of systems would be needed. A useful approach for this would be

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table VI. Filter precision and recall breakdown by model elements.

Model Element	Full	Partial	Missing	Extra
Reset method	100% (1/1)	0%	0%	0%
State variable definitions	100% (1/1)	0%	0%	0%
State variable updates	0%	0%	100% (1/1)	0%
Main execution method	100% (1/1)	0%	0%	0%
State transition methods	100% (2/2)	0%	0%	0%
Transition guard checks (Daikon)	0%	100% (1/1)	0%	0%
Transition guard checks (Custom)	100% (1/1)	0%	0%	0%
Interaction oracles (no output)	100% (1/1)	0%	0%	0%
Interaction oracles (output)	100% (1/1)	0%	0%	0%
Mock object return values	-	-	-	-
Parameter values	0%	0%	100% (1/1)	0%
Return value oracles	-	-	-	-

similar to that which we followed - start from all the provided invariants, analyse which ones are relevant and investigate how these can be effectively (automatically) turned into guard checks. From this we believe more generic and powerful transition guard generation approaches and guidelines could be provided for different types of systems. However, this study is left out of the scope of this paper.

With respect to the tools we used, it can be summarized that having more specific and effective means of FSM and invariant generation would be useful. Both ProM and Daikon were described to have some limitations from our viewpoint, leading to more manual effort in the testing and verification phase. One option would be to extend the models and tools to support the additional information needed. Another option would be to replace those intermediary models by simpler models representing solely the information needed for the generation of the code, and to build our own custom "FSM" and "invariant" inference engines. For example, existing work on mining temporal invariant rules (e.g. [19]) could be used to address the limitations of discovering timing related transitions from the observations. However, in this context it is important to offer possibilities for the user to observe the intermediary models. For example, the usage of ProM permits the user to assess the completeness of the trace, via many different types of models and visualizations.

Although, in most cases, the root-causes for errors reported by the MBT tool are clear, sometimes they are difficult to identify. The cause of an error may be located in the model, or in the SUT. An effective approach for finding these causes is to create a separate test case with a specific testing tool, such as JUnit, based on the generated test case. This separate test case will reveal all the hidden assumptions in data generation, interactions and similar properties, and allow the user to experiment with different settings. A separate test case permits to do more focused analysis of the failure cause. Currently, these tests have to be created manually. However, the information required for their generation is already available in the test case generated by the MBT tool. With this information, the separate test scripts with related data values and other generated input could be automatically generated, saving considerable effort for these difficult debugging cases.

Finally, sometimes, it might be problematic to have an extensive set of test executions available for the SUT. In these cases, an interesting research approach would be to apply input generation mechanisms, such as search-based heuristics [23] for behavior exploration. In this case, the model

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

becomes more of a representation of all the possibilities with the SUT, and less a representation of what is the expected behavior. Thus it requires also the use of automated classification approaches as described in section 3.6.2.

6. CONCLUSIONS

This paper described a a novel way to generate a complete EFSM (including test input, oracles, and harness) out of observations from system executions, suitable for use as an advanced starting point (model) for model-based testing. An important note when using a generated model for software testing and verification is that this model is not a description of what should be *expected* from the SUT, but rather what it *actually* provides. In this regard, one of the main contributions of this paper is also the proposed method that shows how to verify the correctness of the model vs the specification, while using it for testing and verification.

The usefulness of the proposed approach was demonstrated with the help of two real software components, where the generated model was shown to be highly complete, providing an advanced starting point, and where additional test coverage was gained and several new bugs were discovered with the help of the proposed method. We also discussed the limitations of our approach, and proposed means to address them in future work.

Future work will comprise an application of search heuristics, i.e., evolutionary testing techniques, for the generation of test stimuli in order to obtain the traces, automated classification of inputs and outputs captured from the execution scenarios, addressing domain object creation and improvement on the generation of transition guards in order to make them more generic.

REFERENCES

1. Groovy - model-based testing with ModelJUnit. In *http://groovy.codehaus.org/Model-based+testing+using+ModelJUnit*, Referenced August, 2009.
2. W. M. P. v. d. Aalst, B. F. v. Dongen, C. W. Günther, R. S. Mans, A. K. A. d. 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, 2007.
3. W. M. P. v. d. Aalst, V. Rubin, H. M. W. Verbeek, B. F. v. Dongen, E. Kindler, and C. W. Günther. Process mining: A two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling (SoSyM)*, 2009.
4. 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.
5. M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proc. of the Int'l. Symposium on Software Testing and Analysis (ISSTA2006)*, pages 169–179, Portland, Maine, USA, 2006.
6. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.
7. B. Cornelissen, A. Zaidman, A. v. Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 2009.
8. M. d'Amorim, C. Pacheco, D. Marinov, T. Xie, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proc. of the 21st Int'l. Conf. on Automated Software Engineering (ASE'06)*, pages 59–68, Tokyo, Japan, Sept. 2006.
9. S. Ducasse, T. Girba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR'06)*, pages 37–46, 2006.
10. S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Softw. Eng.*, 31(4):312–327, Apr. 2005.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

11. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.
12. H.-G. Gross. *Component-Based Software Testing with UML*. Springer, Heidelberg, 2005.
13. 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.
14. A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proc. conf. of the Centre for Advanced Studies on Collaborative research (CASCON'04)*, pages 42–55, 2004.
15. M. Haran, A. Karr, M. Last, and A. Sanil. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Trans. Softw. Eng.*, 33(5):287–304, 2007. Orso, Alessandro and A. Porter, Adam and Fouche, Sandro.
16. . T. Kanstrén. A study on design for testability in component-based embedded software. In *Proc. 6th Int'l. Conf. on Softw. Eng. Research, Management and Applications (SERA'08)*, pages 31–38, Prague, Czech Republic, 2008.
17. . T. Kanstrén. Program comprehension for user-assited test oracle creation. In *Proc. 4th Int'l. Conf. on Softw. Eng. Advances (ICSEA'09)*, Porto, Portugal, 2009.
18. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.
19. D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *J. Softw. Maint. Evol.*, 20(4):227–247, 2008.
20. D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *Proc. 7th European Softw. Eng. Conf. and the ACM SIGSOFT Symposium on Foundations of Soft. Eng.(ESEC/FSE'09)*, pages 345–354, Amsterdam, The Netherlands, Aug. 2009.
21. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. 30th Int'l. Conf. on Softw. Eng. (ICSE'08)*, pages 501–510, Leipzig, Germany, May 2008.
22. L. Mariani and M. Pezzè. Dynamic detection of COTS component incompatibility. *IEEE Software*, 24(5):76–85, Sept. 2007.
23. P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.
24. A. Mesbah and A. v. Deursen. Invariant-based automatic testing of ajax user interfaces. In *31st Int'l. Conf. on Softw. Eng. (ICSE'09)*, Vancouver, 2009.
25. H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, pages 47–60, 2000.
26. M. L. Nelson. A survey of reverse engineering and program comprehension. *CoRR*, abs/cs/0503068, 2005.
27. C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 –- Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 2005.
28. C. D. Roover, I. Michiels, K. Gybels, and T. D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proc. 14th Int'l. Conf. on Program Comprehension (ICPC'06)*, 2006.
29. D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automated test factoring for java. In *Proc. of the 20th Int'l. Conf. on Automated Softw. Eng. (ASE2005)*, pages 114–123, 2005.
30. S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proc. 7th European Softw. Eng. Conf. and the ACM SIGSOFT Symposium on Foundations of Soft. Eng.(ESEC/FSE'09)*, Amsterdam, The Netherlands, Aug. 2009.
31. N. Tillman and W. Schulte. Mock-object generation with behaviour. In *Proc. of the 21st Int'l. Conf. on Automated Softw. Eng. (ASE2006)*, pages 365–368, Tokyo, Japan, 2006.
32. I. T. Union. Recommendation ITU-R M.1371-1, 2001.
33. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2006.
34. N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe. Automated discovery of state transitions and their functions in source code. *Softw. Test., Verif. Reliab.*, 18(2):99–121, 2008.
35. T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.

Copyright © 2007 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

PAPER VII

Behavior Pattern-Based Model Generation for Model-Based Testing

In: Proceedings of the 1st International Conference on Pervasive Patterns and Applications, PATTERNS 2009, Athens, Greece, November 15–20, 2009. 9 p.
© 2009 IEEE.
Reprinted with permission from the publisher.

Behaviour Pattern-Based Model Generation for Model-Based Testing

Teemu Kanstrén
VTT Technical Research Centre of Finland, Kaitoväylä 1, 90571 Oulu, Finland
& Delft University of Technology, The Netherlands
teemu.kanstren@vtt.fi

***Abstract*—This paper presents the concept of using behavioral pattern mining to generate models for model-based testing. These patterns are mined from observations captured from execution scenarios of the system under test, and the different patterns are combined in order to provide a suitable higher-level model for model-based testing. The concept is first discussed on a general level, providing a basis for implementation of semi-automated model generation algorithms based on combinations of different behavioral patterns. This concept is illustrated by showing how to generate extended finite state-machine models in a format suitable for model-based testing. The generated model is further validated by applying it for model-based testing of a real software component, where it reveals actual faults in the system under test. In addition to the benefits, the discovered limitations of the approach are discussed. Future work is discussed as potential means to address these limitations.**

Keywords-model generation, model-based testing, pattern mining, behavioral patterns.

I. Introduction

Model-based testing (MBT) makes use of models as a basis for generating tests for the system under test (SUT) [1]. When suitable models are available, this can be a powerful approach providing automated generation of test cases for the SUT. However, in many cases suitable models are not available, and their creation and maintenance can be expensive. These models typically need to describe the SUT from the point of view of what is being tested, which can be significantly different from models used for other software development activities. For this reason, they also require special modeling skills in creating models that effectively describe a system for MBT. Since testing is often seen as an activity that provides no extra functionality for the final customer [2], getting the required support and resources for building these specific models can be difficult. This serves to make the adoption of MBT expensive and more difficult, and providing any automated support for creating these models has great potential to make MBT more feasible.

This paper presents the concept of using behavioral pattern mining to generate suitable models for MBT. A basic model is automatically generated, and serves as a starting point to be manually refined and verified with regards to the SUT specification. This is supported by using a MBT tool to execute the model by generating tests and executing these against the implementation, providing a constant feedback-loop for the refinement of the generated model. This provides a semi-automated approach to model creation, making the adoption of MBT easier from the viewpoint of requirements for providing suitable models.

Generating models based on mining behavioral patterns requires first decomposing the target model into suitable behavioral patterns that can be mined from observations of the SUT execution, and combining these patterns to produce a complete model suitable for MBT. In this paper, the information used for pattern mining is observations captured from execution scenarios of the SUT by monitoring its runtime behaviour. This requires the existence of suitable execution scenarios to exercise the SUT to capture a sufficient set of observations, such as a suite of unit tests or data captured from user sessions. This is a common requirement for techniques making use of dynamic analysis (of runtime behaviour) [3], and most existing systems have these types of execution scenarios available.

To demonstrate this approach for model generation, it is applied for generating an extended finite state-machine (EFSM) in a format suitable for MBT. This model is decomposed into suitable behavioral patterns, the information to be captured for mining the required patterns is defined, and means for combining these patterns to generate a basis for the EFSM is given. This is applied to testing an actual software component, revealing actual faults in its implementation. Finally, the experiences including the discovered limitations of the approach are discussed. Possible future directions for addressing these limitations are discussed.

The rest of the paper is structured as follows. Section 2 discusses the related concepts and briefly outlines related work. Section 3 describes the concept of generating models for MBT based on behavioral patterns at a general level. Section 4 discusses the various points of using these generated models for MBT. Section 5 presents a case study of applying a generated model in practice. Section 6 discusses the experiences related with the case study, the limitations of the approach and possible means to address these limitations in future work. Finally, conclusions summarize the paper.

II. Background and Related Work

Many different techniques have been developed that generate state-based behavior models based on behavioral patterns captured from the analysis of software execution scenarios. This section gives an overview of previous research in these related areas.

Daikon is an invariant inference engine used to infer likely invariants based on execution traces [4]. These invariants are described as likely invariants, as they hold for all the observations in the trace, which may or may not contain a representative sample of the SUT behavior. Example inva-

riants include x < 100 (value of x is always observed to be less than 100), and x in Clients (value of x is always observed to be included in the array Clients). Thus these invariants describe behavioral patterns over the data processed by the SUT. In general, the inferred invariants can be described as properties that hold at certain points of the SUT execution [4].

Test generation techniques based on program invariants include Agitator [5], Eclat [6] and the technique proposed by Xie and Notkin [7]. Each provides a tool that generates test input data, and based on the captured execution trace, presents a set of invariants describing the SUT behavior to the user. The user can analyze the proposed invariants to see if the SUT is working according to specification, and turn the invariants into assertions with related test input to form new test cases for the test suite. The approach presented in this paper makes use of these data invariants as one of the behavioral patterns forming an EFSM. This is similar to the previous work in using them to provide expected input- and output-data values with a new application in the MBT domain. In addition, they are combined with state-machine patterns to form parts of a higher level EFSM model for MBT.

Lorenzoli et al. [8] model a system based on captured observations (an execution trace) including method invocations, parameter values, and global state. Similar to the EFSM model generation approach presented in this paper, they use Finite State Machines (FSM) and Daikon-invariants to create the EFSM. The FSM describes behavioral interaction patterns between the SUT method calls, and the invariants patterns of data describe the constraints for the interactions. These EFSM are used for test case selection and test suite optimization with the goal of increasing the coverage of the model. The approach presented in this paper uses similar means to generate the EFSM, but with different algorithms more suitable for MBT, and including the generation of model source code from these models, whereas Lorenzoli et al. generate no tests nor code [8].

A similar approach but in a different domain is presented by Mesbah and van Deursen [9], who build an FSM for web-application user interfaces. In this case the FSM represents interaction patterns in the user interface (UI) of the web-application, and how they affect the UI representation. The composition of UI elements constitutes a new state in the FSM model. Transitions are the clicks (input) to the SUT that caused the UI to change between these states. They use a set of their own invariants, specifically built for web-applications to describe the expected changes in the UI in response to input, as test oracles. These invariants are different from the Daikon provided ones in that they describe the UI elements and their associated state transitions. In the EFSM example used this paper, a similar association of behavioral-patterns related to interactions and processed data is considered from the viewpoint of messages exchanged between components, and matching MBT tools. Another difference is that in this paper, the patterns are provided through pattern mining from the execution scenarios, whereas Mesbah and Deursen expect the checked invariants to be provided by the user [9].

Process mining is a technique developed to mine models for business processes from event logs [10]. Support for process mining has been implemented in a tool called ProM, which can produce various types of models, such as petri-nets and FSM [11] from the event logs. Process mining concepts have also been applied in the software testing domain, to help in validation of service-oriented applications [10]. ProM is used in this paper to provide the behavioral patterns for interaction between components in the form of an FSM. As ProM also supports other types of behavioral models, these could be used in the case where different types of interaction patterns are of interest (such as petri-nets for concurrency).

In order to generate a model for MBT of a SUT, test harness code must also be generated to isolate the SUT from its environment and to verify the correctness of its interactions with the environment. This is commonly achieved with the help of (component) test stubs that emulate the environment. When the stubs are made programmable, they are often referred to as mock objects [12]. This usually means that a component library provides interfaces to create these stubs, and that for each stub it is possible to define the expected interactions with the SUT and the values that should be returned in each case.

Tillmann and Schulte [13], and Saff et al. [14] provide means to automatically generate mock objects for the SUT. Tillmann and Schulte use static analysis (symbolic execution) and Saff et al. use dynamic analysis to capture the expected interaction patterns and return values for the mock objects. Both focus on one test at a time, to allow the generation of mock objects for exactly the purposes of the chosen test. The test for which mock objects are defined is determined by factoring a larger test to smaller tests [14], or based on static analysis of code with symbolic execution [13].

A more specific test harness generation method for service oriented mobile applications is presented by Bertolino et al. [15]. They assume the SUT is described using formal web-service description languages, such as WSDL and WS-Agreement. Based on these specifications, they generate test stubs for components with which the SUT is interacting. WSDL is used to define the stub interfaces, and WS-Agreement to define the expected behaviour of the SUT for the stubs. Additionally, using simulators, they generate data to test the SUT in different situations. This paper uses a similar approach to these mock object techniques for the EFSM example. The captured interaction patterns are used to program the expectations of component interactions for generated tests, and to verify that these correctly happen during test execution.

III. From Behavioral Patterns to MBT Models

This section describes the concept of transforming behavioral patterns captured from the execution scenarios of the SUT into models usable for MBT. The term Model-Based Testing is used here similar to that of Utting and Legeard [1] who describe it as "Generation of test cases with oracles from a behavioral model". The model describes the expected behavior of the SUT, and is used to generate sequences of method invocations and data as SUT stimulus. In order to

validate the correctness of the responses from the SUT, test oracles (as a part of the model) check the expected output data and interaction sequences. The basic elements of an MBT system include the system specification that is used as a basis to create the test model, the test tool used to generate tests based on this model, and the test harness (for online testing) to execute the generated tests against the SUT or test script generator (for offline-testing).

The idea of turning the MBT approach around, and mining the model from the SUT execution scenarios was described by Bertolino et al. [16] as anti-model-based testing, although they never took it further than describing the concept. This approach is the opposite of MBT in the sense that the implementation (the generated model) is compared against the specification instead of manually creating a formal model of the specification and checking it against the implementation.

Using a model for MBT that is generated from existing execution scenarios, such as test suites, presents the question of usefulness of using a MBT tool to generate more tests based on such a model (of existing tests). Although the execution scenarios used as a basis for model generation can include existing test cases, the generated tests can still be useful when the MBT tool generates additional complex interaction sequences based on the combined whole of the underlying patterns. The process of using the generated model also provides means of formal assessment of the implementation against the specification. This is further described in the sections on using the model (Sect. 4) and the case study evaluation (Sect. 5).

The process of model generation described in this paper has two phases. The first phase of model decomposition is generic and needs to be done only once for a single type of model. The second phase of model generation is specific for each SUT. The following subsections describe these different phases in more detail. In the rest of the paper, generating EFSM models is used as an example to illustrate the described concepts. EFSM was chosen as a target model due to many MBT tools supporting this type of a model and its wide application in MBT [1]. Other types of models, such as Petri-nets could be used for different test targets [1] by repeating this process from a different perspective with the chosen model as the target model.

A. *Phase 1:Model Decomposition*

The first phase of model decomposition is illustrated in Figure 1. In the first step of this phase, the target model is defined. In this paper, the target model is the EFSM model. Once this model is defined, the required elements and properties of this model need to be defined. This leads to the decomposition of the model to the behavioral patterns that can be used to generate the complete model from the captured information and mined patterns.

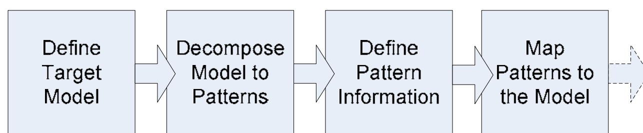


Figure 1. Model decomposition process.

The EFSM model used in this paper requires a representation of *states*, *transitions* and *guards* defining when each of these transitions is allowed to take place. These form the basic elements required for the generation of the EFSM model. In addition to these target model specific properties, also the test automation framework related aspects need to be considered. This means the model needs to be linked to the SUT to provide executable tests (*a test harness*), *input data* for the created tests need to be provided, and the test results need to be verified (*test oracles*). These properties of the target model and the test framework need to be decomposed into behavioral patterns that can be mined from the execution scenarios.

Behavioral interaction patterns can be used to provide the states and interactions by considering the interactions between the system components as state transitions for the EFSM model. When traversing this EFSM, the MBT tool will then generate tests consisting of interaction sequences between the components. There are also two types of interactions, those providing input and those providing output. When generating tests for a SUT, these need to be considered as for each input interaction the MBT tool must generate the input and for each output interaction, the MBT must expect to receive the output (with the help of generated mock objects). Another important property in relation to this is when a given input is generated and when a given output is expected. In different phases of the SUT lifecycle, different inputs produce different outputs. This depends on the given input as well as the internal state the SUT is in.

A basic EFSM representation used in this paper to address these issues is using the input-output transitions as states, and allowing a transition to one of these states when the internal state of the SUT allows for this input-output transition. In this way, each test sequence generated by the MBT is a sequence of expected input-output transitions, depending on the internal state of the SUT. For the test framework related parts, this also requires defining the input values for the input transition and the expected values for the output transition. In order to generate suitable models for this EFSM representation, behavioral patterns need to be defined that can be mined from the observations of execution scenarios for the SUT, and that can be combined to form the target EFSM.

For this purpose, two types of behavioral patterns are used in this paper. The FSM provided by the FSM miner component of the ProM tool is used as a basic representation of the interaction patterns of the components, and the invariants provided by Daikon are used to provide constraints over the SUT internal state and to provide the required input data values and expected output data values. This decomposes the EFSM into two types of behavioral patterns that can be acquired with the help of existing pattern mining tools. Additionally, as described before, the generated tests need to be linked to the SUT to create executable tests (a test harnesss). In the EFSM case this has been achieved by using the input- and output-interface method name definitions as identifiers for the state transitions. This allows linking the generated input- and output-sequences to the SUT methods, providing executable tests, and is an example of linking the

behavioral patterns with additional static information patterns for providing a complete test model. In the next step, the information required to mine these patterns from the execution scenarios needs to be considered.

This definition of required information comes from the decomposed behavioral patterns. For the FSM provided by ProM, the information required includes the messages and their order as passed between the SUT components (for the FSM). For the invariants provided by Daikon, this includes the data describing the SUT internal state at the time of each message call, and the parameter and (possible) return values of each message call. The interface definitions are available from the SUT implementation. The information related to the different model elements for the EFSM case are summarized in Table 1. Once this required information is defined, the tools and algorithms to generate the behavioral patterns from the information, and the final (EFSM) model from the patterns must be defined and implemented.

Table 1. EFSM model decomposition.

Model Element	Pattern	Required Information
State	Data invariants	Data values representing the SUT internal state during each observed (input- and output-) message pass.
Transition	FSM	Input- and output-messages passed through SUT external interfaces.
Transition guard	Data invariants	Input data values for received input-messages, grouped as a separate invariant data point for each input-output message tuple.
Input data	Data invariants	Input data values (e.g. value ranges) used in input messages.
Test harness	Interface definitions	Messages defined in the SUT external input- and output-interfaces.
Test oracles	FSM and data invariants	Output messages (expected interactions) and their data values (expected return values). Associated separately for each separate transition.

The process of mapping the behavioral patterns back to the model is the fourth and final step of the first phase. In the EFSM case, the ProM and Daikon already provide the basic behavioral patterns needed. These basic patterns need to be augmented and processed with specific algorithms to produce the final EFSM model. Defining how this is done completes the fourth step and the first phase. The output from this should be an automated tool that produces the target model from the given pattern information (observations) captured from the SUT execution profiles.

For the EFSM model, as the FSM produced by ProM treats all states and transitions the same, it needs to be augmented with additional information of which messages are input and which are output messages. Each of these is a separate state in the provided FSM and they need to be combined to form a new FSM where the input-output pairs each form their own states. In this way, the input- and output-message sequence pairs will form the basic patterns of expected interactions. This is simple enough by parsing the names of messages from the SUT input- and output-interfaces and associating the FSM states with these input- and output message properties. When the message names are used as identifiers for the ProM event log, this is a straightforward mapping as the names will also match the names of the FSM states. This provides the new FSM where each state describes the expected input-output transitions.

This FSM can now be used as a basis for providing the required states and transitions for an EFSM. However, it still requires the transition guards that define when a transition to a state is allowed to happen and when it is not allowed to happen. With the approach described here, this means that a certain state (an input-output transition) is only allowed to take place when the internal state of the SUT allows the associated input-output transition. For example, consider an example case where a client can request data from a server with a given id value. The server always responds with an output message (transition) but the data in this response depends on the server internal state. If data for the requested id value is available, this is given as a response. However, if data for the id is not available, the response gives an error code. As a second example, consider a SUT that receives messages and relays these to registered listeners for the data it contains. If none are registered, there is no output transition. If any are registered, there are output transitions.

These two types of constraints (transition guards) are not available in a plain FSM. Instead, patterns describing the relations of the internal state of the SUT and the data values in the message parameters are needed to create guards for cases such as requesting data for a certain id. Similarly, patterns describing the relations between the internal state of the SUT and input-output transition sequences are needed, in order to create guards for cases such as expecting notification output messages only in cases where listeners are already registered for the received data. In the EFSM example of this paper, the data invariants provided by Daikon are used to provide this information. These invariants will, for example, say that when no error code is received as output, the request id has been one from the list of connected clients. These invariants can then be turned into transition guards defining that the SUT internal state must match these constraints to allow for this model state (input-output transition) to be explored.

Similarly, the data invariants can be used to define the provided input parameter values. In the example dealing with connected clients, this can be done by forcing a choice of a valid id value from the list of connected clients. Of course, this requires the model to maintain itself a “copy” state of the SUT internal state by keeping, in this case, a list of id values for connected clients it has generated so far (in exploring a state that creates inputs that connect clients).

As can be seen from these examples, the different types of behavioral patterns need to be mapped together to produce the complete target model. When constructing these different patterns (FSM and invariants in the EFSM case), it is important to produce the patterns in such format that they can be mapped together. In the EFSM case, it means that the modified FSM originally produced by ProM needs to be mapped to the data invariants produced by Daikon. To do this, the Daikon patterns are produced using captured data values and invariant identifiers based on the order of messages. Thus, for example, data values for Daikon are recorded based on what was each value when a message was followed by another message. By searching the provided patterns for ones with identifiers matching the input-output pairs of each FSM state, the mapping can be done.

For the EFSM example, this generation of behavioral patterns based on the pattern information and the generation of the target EFSM model based on these patterns has been implemented by the author of this paper in a completely automated tool available as open-source [17]. This tool uses the ProM and Daikon tools to generate the required patterns, creates the intermediate models, and combines these into the format of the used MBT tool.

This subsection has described the model generation using a practical example of generating an EFSM model based on component interactions. Different requirements need to be considered when the different types of patterns are defined that need to be combined for creating the combined model, and these requirements vary depending on the chosen target model and domain. For example, as described earlier in Section 2, a specific model and tool for model-based testing of web-based user-interfaces was presented by Mesbah and van Deursen [9]. They use clicks on the web-page to stimulate the SUT, and expresses expectations as invariants over expected content of the UI elements after provided clicks and input data. To generate this type of model, behavioral patterns are needed to describe the relations of the UI elements, UI navigation commands and input data values. Good candidates again include an FSM with the navigation commands as transitions, UI elements as a state, and data invariants to describe the input data. Similarly, more complex combinations are needed to capture the interactions between data values, navigation commands and how they affect the UI elements of different pages. This shows an example of a different type of an approach, while the implementation details in this case are left as a topic for future work.

The output for this phase should be the definition of the target model, the behavioral patterns it has been decomposed into, the information required to mine these patterns from the SUT execution scenarios, and an automated toolset that mines the required patterns from the provided information and combines these patterns to form the target model.

B. *Phase 2: Model Generation*

The second phase of model generation is illustrated in Figure 2. This phase takes as input from the first phase the information required to mine the behavioral patterns that are used to generate the model, and the mapping of these patterns to the target model as described in the previous subsection. To generate the target model, the information required to mine the patterns needs to be first captured from the execution scenarios of the SUT. As a first step in this phase the SUT must be instrumented to capture the required information. For the EFSM model, the required information was defined as the input- and output-messages of the SUT, the input parameter values and output values, and SUT internal state values. A practical example of instrumenting the SUT to capture this information is presented with the case study in Section 5.

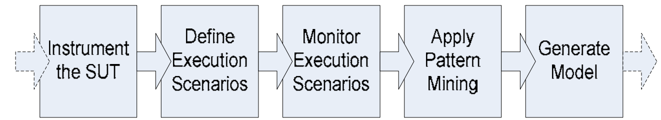


Figure 2. Model generation process.

Once the SUT has been instrumented to capture the required information, it must be executed to capture the actual information from the runtime execution scenarios. A model generated based on information captured from a set of execution scenarios of the SUT is only as complete as the set of scenarios used as a basis. Thus, any behaviour and data values not included in these scenarios are also not included in the mined patterns or in the final model. In order to capture a sufficiently complete model for MBT, the set of scenarios, resulting patterns and final model must be analysed and augmented to form a sufficiently complete set. This is an iterative process of defining scenarios, capturing the information, generating the model, analyzing these and iterating this over until one is satisfied with the set of scenarios. This process and how to perform it effectively is described in more detail in [18]. Capturing the information from the running system is referred to here as monitoring the execution scenarios (step 3).

Once the required information has been captured, tools for mining the required behavioral patterns are applied. In the case of the EFSM model, this means running the ProM and Daikon tools. These tools are given the information in a suitable format for them to process, and as a result they provide the behavioral patterns they were designed to mine. In this case, the FSM model for ProM and the data invariants for Daikon.

When the required behavioral patterns are available, they need to be combined to form the final model. This step is based on the information defined in step 4 of the first phase, where the mapping of the patterns to the model is defined. The previous subsection defined the mapping for EFSM models. In this case, as the required steps are already implemented in an automated tool that generates the patterns and combines them together, this step is a simple application of the tool with the given information (event traces/logs).

IV. USING THE GENERATED MODELS

As described earlier, the process of using the generated models is basically the inverse of the traditional MBT approach. For this reason, using the generated models requires some special consideration. When a model is generated for a SUT based on the information (patterns) mined from chosen

execution scenarios, this model represents only those scenarios and not the generic behaviour of the SUT. It is not a generic representation of all the behaviour of the SUT, but includes only the behaviour and patterns included in the set of used execution scenarios. In this way, the model is only partial and needs to be augmented with additional information to generalize it for testing all the behaviour that should be tested. Also, the generated model does not necessarily describe the correct expected behaviour of the SUT but its actual behaviour as expressed by the execution scenarios. If the implementation is not correct with respect to the specification, this is also reflected in the model. Generating tests with a MBT tool from this type of a model would reveal these differences but simply expect that the model is correct. In order to be useful for testing, this model needs to be compared against the specification to verify the correctness of the implementation.

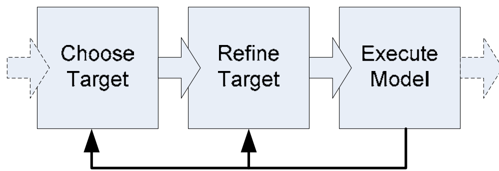


Figure 3. Model refinement process.

The process of using the generated models for MBT is illustrated in Figure 3. This is the manual refinement and validation phase of the model, done with the aid of the SUT specification. Indeed, a computer program (such as a test automation system or a model generator) can not automatically know the requirements and specifications of any given previously unknown SUT. This is similar to the problem of test oracle generation, where verifying the correctness of a SUT is not possible without knowing what should be expected of it (thus something like this would be equal a magical oracle, giving an answer to anything and knowing everything.). Thus a stage where a user checks the correctness of the implementation against its specification is needed.

In this phase, a useful approach is for the user to progress by choosing one target (state) to validate at a time, verifying the generated model parts for this state against the SUT specification, and refining the model to match the specification and the expected behaviour for that state in general. This phase can also be considered as generalizing the model, as the model specific to the used execution scenarios must be refined to match fully the expected behaviour of the SUT as expressed in the specification, not just for those scenarios. As the model is executable with the help of a MBT tool, the refined model can be executed at any time to verify that the implementation still matches the refined model, i.e. that the implementation is correct with respect to the specification.

To progress one state at a time, the different states of the generated model are analysed and their guard statements are refined to match the specification. With the help of these guard statements, it is possible to enable only a part of the states, and focus refinement on these. By enabling more states, the model will also test for more complex behaviour and not single (input-output transition) states. Once the chosen target (state) has been verified and refined, the user can then progress to the next state. Eventually this will lead to a completely verified model, describing all expected behaviour of the SUT from the model viewpoint. One important point in this regard is to note that if behaviour related to some state (input-output transition) is missing from the model, simply checking generated states is not enough alone, but the model must be checked also for missing states, as some may not be implemented in the SUT. This is one type of error that is possible with the implementation vs specification, and was also one of the errors discovered in the case study.

Checking for the expected behaviour here also implies checking that it is indeed the expected behaviour and not just what is implemented. All parts of the generated model need to be checked, including states, transitions, guards, parameter values given for generated test sequences and the assertions checking the expected interactions and return values from the SUT. In other words, the user must check that everything generated from the used execution scenarios also matches the expectations set in the specification. In some cases, this can also highlight missing information or ambiguities in the specification, which also serves to further improve the quality of the SUT by improving its specification and the common understanding about it.

V. A CASE STUDY

This section describes the application of the model generation concepts presented in this paper to a case study of generating a model for a SUT, refining and validating this model and using it to generate tests for the SUT. Only a high-level overview of the case study concepts is given here, for a more detailed description including actual examples of the model code the reader is referred to [19].

The SUT in this case is a component that acts as a database and a server for sensor data. Clients can connect to it, subscribe and query for sensor data with a given sensor id and the server component keeps track of all the information received from different sensors. In this case, the captured execution profile of the SUT (the server component) is turned into a model for the model-based testing (MBT) tool ModelJUnit. This is based on the behavioral patterns provided by the ProM and Daikon tools as described earlier. The model generation process has been implemented in a tool that automatically generates the model from observations captured from the user execution scenarios as described in Section 3.

Two types of execution scenarios were used in the case study, field data captured from actual SUT use and existing unit test cases. This is similar to how execution scenarios are generally constructed in dynamic analysis of running systems [3]. In the case study a set of field data was fed to the application to form one large scale execution scenario, and a set of existing unit tests were used as a set of smaller execution scenarios to cover parts of the behaviour that were not covered by the larger execution scenario.

In order to capture the required information (observations) to be used for model generation, the SUT was instrumented with AspectJ to capture all the messages passed

through its external interfaces. The relevant internal state of the SUT was accessed through a specific interface for this purpose, designed to facilitate its testing. This data was stored in a suitable format for the ProM and Daikon tools to read and to mine their related behavioral patterns. The component to produce the captured pattern information in the correct format for these tools is also implemented as part of the automated model generation tool described in Section 3.

Once the behavioral patterns from ProM and Daikon were available, these were combined together (with the SUT interface descriptions) to form the complete EFSM with the model generation tool. This model was refined part by part against the SUT specification, constantly verifying that each refined part matched the implementation, i.e. that the implementation actually implemented the specification correctly as described in Section 4.

After the model has been fully refined and checked against the specification, six different bugs had been found in the SUT. These bugs were found both through the execution of the refined model as tests generated by the MBT tool, and during the process of refining the model to match the specification. During refinement it was found that there were cases where the specification did not state what should be expected as a return value from a query made to the server with invalid data. The refined model (based on scenarios with only valid input) always expected a certain value, while the implementation would return a different error code. Thus this process also served to highlight refinement needs in the specification. A second type of error found related to checking the refined model against the specification was that of missing implementation for a required behaviour. This was visible as a missing state (intput-output transition) in the model when making a comparison against the specification.

During the execution of the refined model as tests generated by the MBT tool, previously unknown bugs were found due to two main properties of the produced model and its execution. The first property is due to the MBT tool systematically analyzing the model and generating tests to cover more possible complex interaction and input data sequences that were not part of the previously existing tests. The second property is due to the inclusion of systematic asserts to verify all interaction patterns and related data values received as return values, which also checked properties previously considered obviously simple and not checked by the existing tests.

VI. Discussion

This section discusses the experiences and limitations of the presented approach and how these limitations could be addressed in future works. This discussion is based on the results from the presented case study.

Throughout this paper the automated generation of an EFSM model was used as an example. The presented implementation for automatically generating these models makes use of the ProM and Daikon tools for mining the required patterns from the observations made from the execution scenarios. These are generic tools intended for providing either behavioral interaction models (ProM) or data invariants (Daikon). When such tools are available, they are useful in providing a ready component to reuse for the model generation implementation. However, their generality also makes them less useful in generating effective models for a specific domain such as MBT. As the generated patterns are more generic, they require more work in the refinement phase to generalize and make usable for MBT.

In the EFSM case study, the data invariants provided by Daikon were first used as such to provide the transition guards for the states. This provided both too many (useless) invariants, and the invariants that were useful were overly constrained. This is due to Daikon being a tool intended to provide generic invariants over the data values. For this reason, it produces all possible invariants it can find although most of them are not useful from the MBT and EFSM transition guard viewpoint. For example, in the EFSM case it provided invariants describing the relations of the size of the internal state variables to each other, and the contents of a state variable array always being constant. Only the ones related to the size of the internal state variables alone (not in relation to other state variables) were found useful, and the rest had to be discarded to remove the useless invariants (and generated guard conditions).

A second limitation is due to the Daikon invariants being limited to the data provided by the execution scenarios. For example, when global state is represented in the form of a list, and the size of the list is either 1 or 3 in the execution scenarios, a generic approach gives a condition that the value must always be either 1 or 3. In this case, a more optimistic assumption had to be made and a guard condition was generated to require that this list always contains some items (size > 0).

These limitations of the Daikon invariants were addressed in the EFSM case study by implementing the described, more optimistic, approaches as custom algorithms to further process the Daikon invariants from the MBT model generation viewpoint. Whereas using the invariants directly initially provided weaker results, this implemented abstraction proved to provide very powerful generation of the transition guard of the model. This shows how the basic patterns need to be analysed and applied from a more domain specific viewpoint for useful results. However, identifying a good set of candidates and making them more specific for the domain and the targeted model requires more extensive studies with various components, state representations and input data sets, or similar properties of the target model. Thus it is also dependant on the target model, the state representation of the SUT, and similar properties.

Maintaining state inside the test model is another area where the use of invariants could have been improved. Invariant detection could be extended to automatically cover both pre- and post-conditions in the form of providing invariants over the SUT internal state values, both before and after a message is processed. For nor now, the implementation is focused on the pre-conditions, which means that, for example, it is not possible to infer an invariant stating whether a parameter value should become a part of a state expression (such as a list variable, or a UI element), after a state transition. This is again mostly due to difficulties in making a tool built for specific, more generic cases, adapt to these special

requirements. In fact, this can be seen as a requirement for a different type of a behavioral pattern, related to the relations of the values in the other patterns over time.

Concerning the FSM code generation, a set of issues were encountered with regards to multiplicity of state transitions and the abstraction provided by the FSM. ProM provides a generic FSM from the observations, where it is not possible to say whether a single output, multiple outputs or sometimes no output at all follows an input or only in some cases. It only tells that it follows, if the output is there in any given scenario. Since it is not possible to infer from the provided FSM what are the expected combinations, all combinations must be generated and the user must remove the excess ones during model refinement. Again, a specific FSM miner for the purposes of MBT model generation could preserve this information, allowing for automatically leaving out the excess states.

With respect to the tools used, it can be summarized that having more specific and effective means of mining the required patterns from the perspective of the target domain and model and its use in MBT would be useful. Either by extending the general tools with more specific extensions or as separate tools specifically built for this purpose. In this regard, it is also important to consider the ability to inspect the generated patterns and models during different phases as described in [18]. Taking all these requirements into account, the effort to build such tools especially for the purposes of a specific model is not trivial, and as such a more generic approach to produce different models through extensions of a basic framework architecture has more potential.

A second case is that of debugging the root-causes of the found failures. These can be complex to analyse and pinpoint to cause of failure into problems with the refined model or the implementation. One effective approach for finding these causes is to create a separate test script from the test generated by the MBT. This separate test case will reveal all the hidden assumptions in data generation, interactions and similar properties, and allow the user to experiment with different settings. Currently, these tests have to be created manually. However, the information required for their generation is already available in the test case generated by the MBT tool. With this information, the separate test scripts with related data values and other generated input could be automatically generated, saving considerable effort for these difficult to debug cases.

Although not largely discussed in this paper, test oracles can also be created based on the behavioral patterns. This requires being able to make classifications of the patterns to those that should be expected or not. This is another topic to consider in the context of the supporting tools and is discussed in more detail in [18].

Finally, sometimes, an extensive set of execution scenarios is not available for the SUT to be used as a basis for making the required observations. In these cases, techniques such as automated test data generation could be used to automatically generate execution scenarios. However, this can easily lead to the generated model containing extra “noise” in the form of various different types of data values and interactions exercised. Thus it would also require more effective methods for inferring the most interesting patterns correctly, such as more specific pattern mining tools as discussed before in this section.

VII. Conclusions

This paper described the concept of using behavioral pattern mining to generate models for model-based testing. The described process consists of choosing the target model, decomposing this model into a set of behavioral patterns, defining the required information to mine these patterns from a set of observations made from running the execution scenarios for the SUT. The process of turning these mined patterns into the suitable target models for model-based testing (MBT) was described. As the concept of using generated models as a basis for MBT is opposite of the traditional approach to MBT where models are usually created from the specification, the use of these models in the context of MBT was also discussed and a process for this was presented.

Throughout the paper, the generation of an extended finite state-machine model suitable for MBT was used as an example. A practical implementation of an automated tool for this was presented and the whole approach was validated with a case study of its application, where it revealed faults in the implementation of a real software component. In addition to describing the concept and its validation, the limitations of the approach as based on the experiences from the case study were discussed. These limitations form a basis for improving the presented methods in the context of future work.

Acknowledgment

The author wishes to thank Eric Verbeek for his help with the ProM tool, and Eric Piel and Hans-Gerhard Gross for providing the case study environment and discussions and comments related to the topics discussed in this paper. This work has been supported by the Nokia foundation.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*.: Morgan Kaufmann, 2007.

[2] T. Kanstrén, "A Study on Design for Testability in Component-Based Embedded Software," in *Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications*, Prague, Chezh Republic, 2008, pp. 31-38.

[3] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transaction on Software Eng.*, 2009.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Eng.*, vol. 27, no. 2, pp. 99-123, Feb. 2001.

[5] M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," in *Proc. Int'l. Symposium on Software Testing and Analysis (ISSTA'06)*, Portland, Maine, 2006, pp. 169-179.

[6] C. Pacheso and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *Proc. European Conf. on Object-Oriented Programming (ECOOP'05)*, 2005, pp. 504-527.

[7] T. Xie and D. Notkin, "Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions," *Journal of Automated Software Engineering*, vol. 13, no. 3, pp. 345-371, July 2006.

[8] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Proc. 30th Int'l. Conf. on Software Eng. (ICSE'08)*, Leipzig, Germany, 2008, pp. 501-510.

[9] A. Mesbah and A. van Deursen, "Invariant-Based Testing of Ajax User Interfaces," in *Proc. 31st Int'l. Conf. on Software Eng.*, Vancouver, Canada, 2009.

[10] W.M.P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A. K. Alves 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 *Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN07)*, Siedlce, Poland, 2007.

[11] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther, "Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting," *Software and Systems Modeling (SoSyM)*, 2009.

[12] T. Mackinnon, S. Freeman, and P. Craig, "Endo-Testing: Unit Testing with Mock Objects," in *Proc. eXtreme Programming and Flexible Processes in Software Eng. (XP2000)*, Cagliari, Sardinia, Italy, 2000.

[13] N. Tillmann and W. Schulte, "Mock-Object Generation with Behaviour," in *Proceedings of the 21st IEEE/ACM Internation Conference on Automated Software Engineering*, Tokyo, Japan, 2006, pp. 365-368.

[14] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automated Test Factoring for Java," in *Proc. 20th ACM/IEEE International Conference on Automated Software Engineering (ASE'05)*, 2005, pp. 114-123.

[15] A. Bertolino, G. De Angelis, F. Lonetti, and A. Sabetta, "Let the Puppets Move! Automated Testbed Generation for Service-Oriented Mobile Applications," in *Proc. 34th Euromicro Conf. on Software Eng. and Advanced Applications*, Parma, Italy, 2008, pp. 321-328.

[16] A Bertolino, A Polini, P Inverardi, and H Muccini, "Towards Anti-Model-Based-Testing," in *Fast Abstracts in International Conference on Dependable Systems and Networks (DSN'04)*, Florence, 2004.

[17] Framework for Dynamic Analysis and Test. [Online]. http://sourceforge.net/projects/noen/, referenced August 2009.

[18] T. Kanstrén, "Program Comprehension for User-Assisted Test Oracle Generation," in *Proc. 4th Int'l. Conf. on Software. Eng. Advances*, Porto, Portugal, 2009.

[19] T. Kanstrén, E. Piel, and H-G. Gross, "Trace-based code generation for model-based testing," in *Technical Report TUD-SERG-2009-017, Delft University of Technology, Software Engineering Research Group*, 2009.

PAPER VIII

Program Comprehension for User-Assisted Test Oracle Generation

Proceedings of the 4th International Conference on Software Engineering Advances, ICSEA 2009, Porto, Portugal, September 20–25, 2009. 10 p.
© 2009 IEEE.
Reprinted with permission from the publisher.

Program Comprehension for User-Assisted Test Oracle Generation

Teemu Kanstrén
VTT Technical Research Centre of Finland
Kaitoväylä 1, 90571 Oulu, Finland
e-mail: teemu.kanstren@vtt.fi

Abstract—**Software testing requires a test oracle that makes an assessment of the correctness of the tested program behaviour, based on a priori created model. While test automation is a popular research topic, there is only a limited amount of work in the subject of automating the process of creating test oracles. This lack of test oracle automation greatly limits the usefulness of automated testing techniques. One reason for this is the difficulty to automatically determine the correctness of previously unknown software. Instead the task of coming up with a useful oracle is often left to the user as a manual task. Program comprehension techniques are focused on supporting the building of human understanding for a previously unknown program, and as such are good candidates to assist in the test oracle creation process. This paper addresses the lack of automated support for test oracle creation by providing a framework for using program comprehension techniques to provide automated assistance to the user in creating test oracles. Based on analysis of existing work and theoretical background, the basic concept for this process is defined. A case example demonstrates the practical application of this concept with the generation of a model, including a test oracle, for model-based testing. From the existing approaches and the presented case example, a framework for this type of process is presented in order to provide a basis for providing more powerful techniques for user-assisted test oracle generation.**

Keywords- Test oracle; Program comprehension; Test automation

I. INTRODUCTION

Test automation in software engineering often does not live up to its name and promise. Commonly the test automation is actually a set of test scripts written manually and executed over and over by a tool designed for this purpose. This is useful for regression testing but does not deliver on the promise of automated testing, where one could just run a tool to generate tests for a given piece of software without having to manually create them. A truly automated testing platform would need to automatically generate message sequences to drive the system under test (SUT) through its interfaces, test input data for these messages, a test harness to isolate the SUT from its environment and a test oracle to verify the correctness of the SUT output in response to the input messages and data.

With statements on how software testing takes more than 50% of the total development costs [1], test automation has of course been a popular research topic and numerous research papers have been published related to the automation of different test automation components. Especially test input generation has been a popular research area. One of the least automated parts of test automation remains to be the creation of test oracles. This can be seen to be partly due to the difficulties to automatically (like a magical oracle) determine the correctness of previously unknown software. The specification of what is to be expected of a SUT comes from its stakeholders, and no program can guess what is expected from another program without external input. Some generic properties of the correct functionality can be devised for specific cases and domains (e.g. refactoring engines [2] and protocols errors in web applications [3]), but the truly automated parts of these are limited and do not generalize.

This paper views the automatically assisted oracle creation problem as an application of program comprehension (PC). Similar topics have been considered before, for example, Sneed has discussed how the human tester is the person who needs to have the best understanding of the SUT [4]. PC is aimed at building a human understanding of software (SW) systems. Research in this field can be classified to study either the human view of cognitive processes used to understand programs or the technological view of building semi-automated tool support for program comprehension [5]. The end result is typically a model describing the program at a chosen abstraction level and from a chosen viewpoint. Finally this model needs to be validated to ensure correct understanding. This is closely related to how a test oracle works, by comparing a model of the expected SUT behaviour against a model of the actual SUT behaviour and verifying that they match.

The focus of this paper is on creating test oracles for existing systems, with the help of dynamic analysis techniques. In the spirit of PC, the goal is not to achieve fully automated generation of test oracles for any SUT but to provide a framework for how automated assistance for creating the test oracles can be provided for the human user. Starting with a theoretical analysis of the concepts, existing approaches for the subjects are reviewed. Next, an example case of applying PC concepts for the generation of a model, including a test oracle, for model-based testing is presented. Finally, existing approaches are summed up together in respect to the presented theoretical background to provide a framework for providing techniques to support test oracle automation with the help of PC techniques.

The rest of the paper is structured as follows. Section 2 provides a general overview of the test oracle concept and existing techniques to support test oracle automation. Section 3 presents the general concepts of PC, and a brief overview of related techniques. Section 4 provides a model describing

the relationship between test oracles and PC, including a comparison of the test oracle and PC techniques presented in the previous sections. Section 5 presents an example case of a PC approach to provide automated assistance for creating test oracles for an existing system. Section 6 discusses the case study and its relation to existing work shown in section 4, analyzing these different concepts and presenting a framework for user-assisted test oracle generation with the help of PC techniques. Finally, conclusions end the paper.

II. TEST ORACLES

This section presents an overview on test oracle concepts and related research to provide a basis for analyzing the synergies to program comprehension in later sections.

A. General Concepts

The concepts of test oracles in this paper follow the definitions used in [6]. A *test oracle* is defined as a mechanism for determining the correctness of the behaviour of software during (test) execution. The oracle is divided into the *oracle information*, specifying what constitutes the correct behaviour, and the *oracle procedure*, which is the algorithm verifying the test results against the oracle information. Further terms are also used according to [6]. Successful test evaluation requires capturing information about the running system using a *test monitor.* For simple systems, it can be enough for the test monitor to just capture the output of the system. For more complex systems, such as reactive systems, more detailed information, such as internal events, timing information, stimuli and responses, need to be captured. All the information captured by the test monitor is called the *execution profile* (EP) of the system, and includes control and data information.

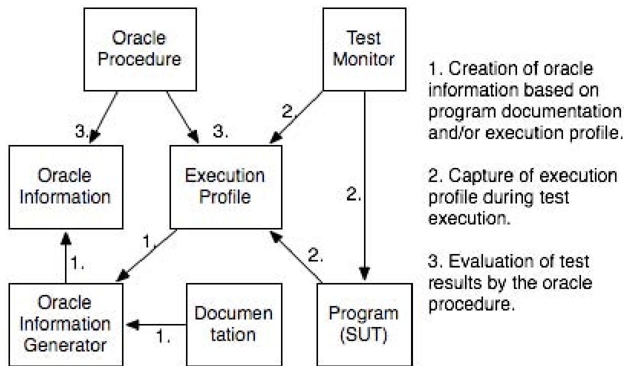


Figure 1. Test oracle components.

The different components of test oracles and their relations are illustrated in Figure 1. These are grouped to three main steps based on the order of their application. Before a test case can be created, the oracle information and procedure need to be defined, which forms the first step. This is typically based on the program specifications and/or EP (in which case step 2 would precede step 1). This information can be generated by a human developer/tester, a test automation program or a combination of both (program supporting a human). In the second step, the test case is executed and the test monitor captures the EP of the system. The input to the EP is the data captured from the SUT execution by the test monitor. In the third and final step, the oracle procedure gives a verdict on test results by comparing the EP to the expected correct behaviour as expressed by the oracle information.

The following subsections give an overview of existing automation techniques to assist in test oracle creation, including both fully automated test oracles (as provided oracle components for a given domain) and automation tools to assist a user in oracle creation.

B. Automatic Test Oracle Components

In the context of automated test oracles, Daniel et al. [2] have presented a set of automated test oracles for refactoring engines. These oracles are based on the properties of the refactoring operations supported, including checking for the invertibility of the refactoring operation (performing the operation backwards to check it produces the original result), and checking that the moving of an element actually results in creating the item in a new location. This type of a test oracle is applicable to different refactoring engines, and Daniel et al. describe applying it on the Netbeans and Eclipse IDE's.

A generic approach for a test oracle is checking for thrown exceptions and application crashes [7]. A more domain specific but similar approach is presented by Mesbah and Deursen [3], who use invariants to define a set of automated test oracles for AJAX-based web-applications. They provide a set of invariant-based test oracles for generic properties of this type of web-applications, such as the HTML always being valid, and the DOM-tree not containing any (HTTP) error messages. These test oracles are then applicable to any AJAX web-application. As an oracle procedure they use an automated input-generation designed to crawl through web-pages and check that the resulting documents do not violate these invariants.

Memon and Xie [8] present an automated test oracle for GUI testing. This test oracle follows the traditional record/replay approach, where the properties of the GUI elements are used to describe its states. A model of the SUT behaviour is captured using a set of existing test cases that are assumed to describe the correct behaviour of the SUT, and a GUI state extraction technique. This model is then used as a basis for regression testing to describe the expected states.

Machine learning has been applied in several studies to generate a test oracle based on the EP of the SUT. These oracles are typically based on low-level EP data, such as capturing all function calls inside a program, their parameter values and the relations of these values [9]. A learning algorithm is trained with EP's labeled as failing and correct, which provides a test oracle that can classify a new execution as passing or failing. These oracles can be more generic than the previous approaches, but typically they need to be trained separately for each SUT and cannot test for any application specific behaviour, such as correct input-output transitions.

These examples summarize the type of support that current automated test oracle components can provide. In case

of Daniel et al. [2] the test oracles can be applied to different refactoring engines, but not to any other type of SW. More generic approaches are based on generic errors or exceptions thrown by the programming language constructs [7][9] or available in domain specific representations [3]. The Memon and Xie [8] approach is mainly applicable to regression testing only, with the assumption that the recorded model is correct, and the model can be fragile with regards to small changes in the SUT behaviour that are irrelevant from the test oracle perspective.

C. User Assisted Test Oracle Automation

A second category in automated test oracle creation support is in user assisted test oracle creation. Typically in these cases, the oracle procedure is provided and the user has to provide the oracle information. Usually, a basis for describing the oracle information is also provided in the form of tools or libraries that can be used to create or describe it.

In addition to providing automated oracle components (with both procedure and information) for the generic properties of web-applications as described earlier, Mesbah and van Deursen [3] also give the user the option to provide custom invariants to be checked, such as the contents of a table being update when a link is clicked. Their toolset will then automatically crawl through the web-application and check that the provided invariants are not violated. Here the invariants are the oracle information provided by the user and the checking of the invariants is the automated oracle procedure that is given. The toolset also provides means to describe the invariants, and in this way supports the creation of the oracle information.

Andrews and Zhang [10] have presented a technique for test oracle generation based on log file analysis. This is based the SUT writing a log file based using a predefined logging policy and a log file analyser asserting the correctness of the execution based on the log file. Their approach requires writing the log file analyser component and providing a matching logging policy to support the analyser. They illustrate the approach with a state-machine based matching, where the transitions are based on the available log lines. The log file analyser component is applied against log files collected from SUT execution, and makes the assertion whether the log file matches the expected behaviour or not. In this case the oracle information is provided in form of the analyser component that the user has to write. This is supported by the logging policy and the interfaces to their test execution system (the oracle procedure).

Both Ducasse et al. [11] and De Roover et al. [12] have described similar techniques for building test cases based on traces collected from a programs execution. They start with executing the SUT and collecting traces from the execution. Logic languages derived from Prolog are used to query the execution traces, and these queries act as the test oracles. The queries assert that the recorded behaviour matches the expected behaviour. Ducasse et al. [11] use the queries to filter relevant data from large, low-level, data sets, while De Roover et al. [12] do similar queries but aim at limiting the trace data to higher level events and lighter trace implementation. The aim with these techniques is to produce a model that is both human understandable and machine verifiable, in order to support both test automation and PC. In this case, the user has to provide the oracle information in the form of a query that describes what should be found in the (EP) trace. The oracle procedure is the test automation system that executes the queries and reports their results, doing a comparison against set expectations.

Lienhard et al. [13] describe the use of a visualization technique, based on a program execution trace, as a basis to assist the user in creating unit tests. This visualization is called the Test Blueprint. They focus their analysis on a part of a program execution, which they term an execution unit, in order to reveal so called side-effects. These side-effects describe the created and changed objects, changed object references and similar properties during the execution of the chosen unit. These are then provided to the user through their visualization, which helps the user in turning them into assertions. These assertions are used to verify that no important properties (side-effects) are violated during changes of the SUT. They also describe the visualization as supporting the creation of a test harness, as it shows required interactions with other objects. Creating the assertions in this case is done manually.

Program invariants are used as a basis for assisted oracle-generation in Agitator [14], Eclat [15] and in the technique proposed by Xie and Notkin [16]. These techniques require a set of program executions as a basis (such as existing test cases or an example program) and based on this create an invariant model to describe the SUT. This model is based on capturing all method calls and their parameter values (the execution profile). They then provide the user with the option of turning these invariants into assertions as part of the SUT unit test suite, to check that the invariants are not violated in regression testing. In this case, the user is actually provided with a form of a test oracle procedure and information. The oracle procedure is the assertion facility of the used unit test tool, and the oracle information is the invariants that are suggested to be turned into assertions. The procedure is not fully automatic, it requires the user to evaluate the usefulness of the proposed invariants, augment or modify them where needed and to choose which ones should be turned into assertions. As such, they can be provide highly automated support but, due to focusing on low-level execution data, are limited in their usefulness (much like the machine learning approaches described earlier) to unit- tests of small granularity classes or components. Higher level concepts, such as the properties of input-output transitions in relation to specification are not supported as such oracle information is in practice not embedded in program structure for invariant detection.

III. Program Comprehension

This section provides an overview of program comprehension concepts and related research to provide a basis for analysing the synergies with test oracles in later sections.

A. General Concepts

Program comprehension is a field dealing with human understanding of software systems, and its theoretical foun-

dations are based on fields studying human learning and understanding. The theories of PC are also referred to as cognitive theories of program comprehension [5], which highlights the purpose of PC techniques and tools as an aid to building human understanding of software systems. Program comprehension can make use of information from different sources, such as static analysis of program artefacts (e.g. source code) and dynamic analysis of program executions. In the context of this paper, when PC is discussed, it refers to techniques related to dynamic analysis.

Figure 2 shows the process of PC as a three-step process based on [17], adapted to include the impact of analysis of the program itself on the hypothesis of the program purpose. First, based on the program documentation, a hypothesis is made for what is the purpose of the program and how it is expected to work. This step can also be influenced by analysis of the program itself when the documentation is not up-to-date. The program is examined in the second step to build a hypothesis on how it operates. The building of this hypothesis can be influenced by the hypothesis on the program purpose. Finally, it is attempted to match the two hypotheses together to see if the understanding is correct. If this step fails, it forces a return to either step one or step two.

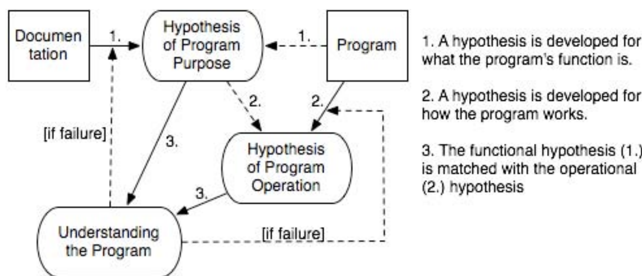


Figure 2. Program comprehension process.

In order to provide the required basis for how the application of PC techniques for test automation is demonstrated in the case study section later, the basic approaches for PC need to be reviewed. Two basic approaches to program comprehension are typically identified: top-down comprehension and bottom-up comprehension [5]. Both can be mapped to the process described in Figure 2, where the top down emphasizes documentation for step 1 and bottom-up emphasizes the program part. Further, from these a hybrid model called an integrated metamodel has been presented [18]. Other types of models include a knowledge based model, opportunistic and systematic strategies, and consideration of program and programmer characteristics [5].

The bottom-up approaches assume that the comprehension process starts from low-level concepts, such as reading source code statements, and grouping these into higher-level concepts [19]. In these models the programmer starts from composing small chunks to progressively larger chunks, finally acquiring a model for the program or its parts under investigation.

The top-down approach was presented by Brooks [20], who describes program comprehension as building knowledge about the problem domain and mapping it to the program source code. First an initial hypothesis is formulated based on the programmer's knowledge about the program domain. Based on information extracted from the program, the hypothesis is refined and subsidiary hypothesis can be generated. The verification of these hypotheses is based on beacons, which are described as sets of features (details) in the code that typically indicate the occurrence of certain structures or operations related to the hypothesis. It is seen that the investigation of a program will identify strong beacons for all hypotheses, and these beacons will lead to further refinement of the hypotheses.

In a hybrid approach, the programmer is seen to switch between these top-down and bottom-up approaches as seen necessary and as the analysis of the program progresses [18]. One is seen to move from the specification to source code and use all these available information sources as needed, and as described in the top-down and bottom-up approaches.

B. Existing Techniques

This subsection gives a brief overview of existing techniques related to PC with dynamic analysis. Since the intent is to provide a basis for mapping from PC to test oracle automation, the focus is on behavioral models as this allows matching them against test oracle requirements. As PC is a field with a large number of techniques and studies [21], the focus is only to give an overview of this area.

In order to support the human cognitive process of program comprehension, various tools and techniques have been presented with different approaches, and aiming different properties of the SW, such as structure and behaviour. These approaches include visualizations, pattern detection, summarization (e.g. clustering), data queries, and filtering or slicing the data [21].

Sequence diagrams are a popular means to model the behaviour of the analysed system [22][23]. These tools are intended to support functions such as mapping sequences of messages (method invocations) to features of the analysed SW (top-down approach), and to understand patterns of execution (bottom-up approach) [22].

State machines are another popular means of modeling SW behaviour, and many approaches to synthesize state-machines based on execution traces have been presented (e.g. [24], [25]). Although many of these list the support for PC as one of the uses, there are only few studies on actual use and how the generated state-machines assist humans in PC [21]. Although it is intuitive that understanding the states of a system and how the transitions between them happen help in PC, it would be useful to see empirical studies on how people actually use them to support this process.

Other types of models include invariant models [26], concurrency models [27], architectural models of components and connectors [28], and petri-nets [29]. These are described to help in tasks such as understanding the concur-

rency related dependencies [27][29] or structures of the system [28]. However, these too suffer from lack of empirical studies on how they would be systematically applied by their users. This lack of empirical studies (controlled experiments) on how the human users make use of these models is also highlighted as one of the lacking areas in PC research by Cornelissen et al. [21].

IV. TEST ORACLES AND PROGRAM COMPREHENSION

This section presents an analysis of the relations between the test oracle and program comprehension concepts presented in the previous two sections, and related research.

A. *General Concepts*

The previous two sections presented the basic concepts of test oracles and program comprehension and an overview of research done in these areas. This section reviews these two concepts together from the viewpoints of commonalities to use as a basis for finding synergies between them. Figure 1 and Figure 2 showed an overview of test oracle and PC procedures accordingly. Figure 3 shows these two figures at a higher abstraction level and maps them together. The top row shows the test oracle process and the bottom row shows the PC process.

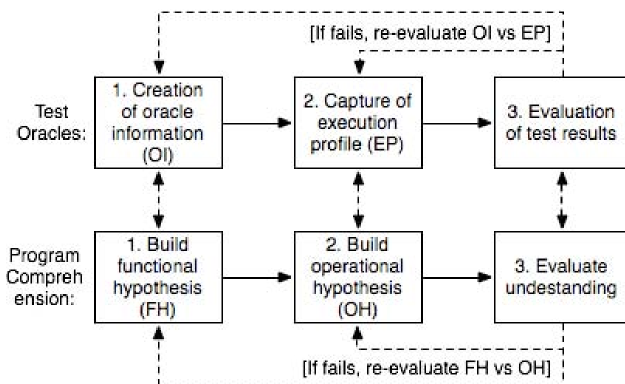


Figure 3. Test oracles and program comprehension.

Figure 3 shows how the two concepts are similar by mapping each of the three steps in both processes to each other and also providing a similar feedback loop in both processes. The details of these steps were described in the previous sections and are not repeated here. Instead this section is focused on discussing the conceptual similarities of these two processes.

In the first step, the creation of oracle information (OI) uses program documentation and execution profiles as input to create a specification of what is expected from the SUT. Similarly, in the first step of the PC process a functional hypothesis (FH) is built for the program based on its documentation and artefacts such as the EP. Thus, both processes have conceptually similar inputs, outputs and goals in this step.

In the second step of the oracle process, the EP of the SUT is captured. This EP describes the behavior of the SUT implementation. In step 2 of the PC process, the operational hypothesis (OH) is built to describe how the program operates. Again, both these processes have similar goals, inputs and outputs in this step. Both aim at building a model to describe what the program is and what it does. Both also use the program and its executions as input.

In the third step, the oracle process compares the OI model against the EP model to evaluate the test results. If these are found to match, the test result is marked as passed; otherwise it is marked as a failure. Similarly, in the program comprehension process, this step involves evaluating the FH against the OH. If these are found to match, the program comprehension is seen to be successful; otherwise it is seen to have failed.

Finally, in both processes, a failure in the third step prompts a return to the earlier model generation phases. As the evaluation is in both cases based on comparison of two models (or hypothesis), this step leads to the re-evaluation of both of these models to see which one(s) are not correct, refining these models based on this evaluation and repeating the process.

B. *Existing Techniques*

Many techniques that are mainly aiming to model a SW system list a number of possible fields where the authors think models generated from execution traces could be used. These fields usually include both PC and software testing. However, more concrete evaluations for all the included uses are in many cases missing, as usually a paper can only have one effective focus area. Despite this lack of studies, it is true that generated models at a higher level of abstraction than pure execution trace represented by function calls and parameter values are of course easier to understand for humans. As they also describe the executions of the SUT, they can be considered to have possible uses for software testing.

Some of the better examples are found in the research described in the earlier section about user assisted test oracles. For example, both Ducasse et al. [11] and De Roover et al. [12] describe their techniques as supporting both software testing and PC. They describe their tools from the software testing viewpoint, in the form that enables the user to create queries over the SUT traces and once satisfied with the answers, to turn these into assertion for the test suite of the SUT. PC is seen to be supported in answering the queries the user has about the program, and test automation in keeping these queries as a part of the regression test suite. As a part of the regression test suite, they can also be seen as upholding that understanding by reporting when the related assumption no longer holds.

Similarly, the previously describe work by Lienhard et al. [13] on their Test Blueprints technique aims to support the creation of test oracles with the aid of program comprehension techniques. The visualization they use is originally developed to support program comprehension, and in this case they also use it to help the user understand the SUT in order to create test assertions, which act as test oracles.

Invariant detection started out from work on producing models based on execution profiles and was described as potentially supporting many different domains, such as test automation and program comprehension [26]. A number of tools including Agitar [14], Eclat [15] and the technique

presented by Xie and Notkin [16] use these invariant based models as a basis and are described as supporting both software testing and PC. They exercise the SUT with a set of scenarios, either with generated input or with existing test cases. The inferred invariants of the SUT are then presented to the user and the user is given the option to turn these invariants into assertions to create new test cases along with related input data. PC is supported by describing the SUT behaviour as a set of invariants, test automation in allowing the user to turn these invariants into assertions for the test suite. Again, PC can also be seen to be supported by the inclusion in a regression test suite, which results in reporting when these assumptions no longer hold.

V. A CASE EXAMPLE

This section presents a case example of applying the concepts presented in this paper. Previously the theoretical background for the use of PC for user-assisted test oracle creation has been presented, including a brief description of a set of existing tools that can be seen to help in this process. This section is intended to present an example case of applying the previously presented theoretical concepts in practice and is thus termed a case example. With the help of PC tools and techniques, a model, including a test oracle, is generated based on the execution profile (traces) of the SUT. The concepts presented on the previous sections on the background theory are mapped to this concrete example, and it is shown how the PC concepts can help in the different parts of this process.

The SUT in this case is a component that acts as a database and a server for sensor data. Clients can connect to it, subscribe and query for sensor data and the server component keeps track of all the information received from different sensors. In this case, the captured execution profile of the SUT (the server component) is turned into a model for the model-based testing (MBT) tool ModelJUnit[1]. This means the SUT has to be modeled as an extended finite state-machine (EFSM), where it makes transitions from one state to another based on a set of guard constraints that describe when each transition can be taken. The model generation process has been implemented in a tool that automatically generates the model from the execution profile traces.

To start with, when a model is used as a basis for MBT of a SUT, the model should provide a comprehensive description of the SUT at the chosen abstraction level to ensure also comprehensive test generation from this model. As here the model is generated based on the execution profile captured from the existing SUT with a set of defined executions, the completeness of the model depends on the completeness of the execution profile.

As described by Cornelissen et al. [21], PC with dynamic analysis has two typical possible data sources to be used as a basis to analyse a system. One is the existing test suite and the other is any existing sample execution of the program, such as user sessions or example applications. In this case example, first the execution profile of the SUT has been captured by using an example application intended to demonstrate the use of the component. This application feeds about 20000 messages captured from real-world sensors through the component and sets up a set of test clients to illustrate the use of the data.

To check that the execution profile contains a comprehensive description of the SUT to be used as a basis for an EFSM, the execution profile is visualized with the ProM[2] process mining tool. This tool is intended to help people build an understanding of business processes by visualizing event logs as different types of models. By using it to visualize the execution profile trace as an FSM, the completeness of the execution profile can be checked against its specification (in this case written in technical English). This visualization is shown in Figure 4.

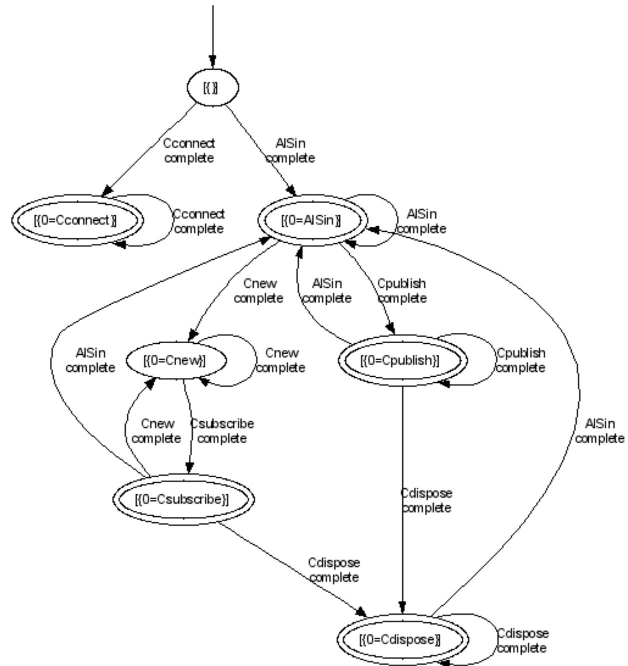


Figure 4. ProM visualization of the initial execution profile.

This visualization and analysis approach can be seen as a bottom-up approach to PC, providing means to gain an understanding of the available SUT execution profile. If only this view is used, it is not possible to tell if the execution profile is a comprehensive representation of the program behaviour for the purposes of testing. Instead, a hybrid strategy needs to be applied and the produced model compared against the SUT specification. From this it was seen that the execution profile was missing some important states and transitions, such as the ability to request sensor data and get a data message back as a reply. To address these issues, the execution profile was augmented with additional focused test cases (executions) intended to capture the missing behavioral states and transitions of the SUT. The resulting model for the execution profile with the example application and six focused tests is shown in Figure 5. This was judged to provide a comprehensive description of the SUT behaviour.

[1] http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

[2] http://www.processmining.org

With a satisfactory execution profile (describing required states) now available, it is used as a basis for generating the model in a format usable by the MBT tool. This shows another benefit of applying the synergies between PC and test automation as described in previous sections. At this point the tool support to build the basic state-machine for the test model is already available in the PC tool and can be reused for test model generation also. ProM is originally built for visualization to support analysis of event logs. However, in recent versions support for using the analysis algorithms from outside its GUI have also been added. Using this support, the test model generation tool implemented by the author of this paper creates the state-machine shown in Figure 5 and uses it as a basis in its own algorithms to generate the test model.

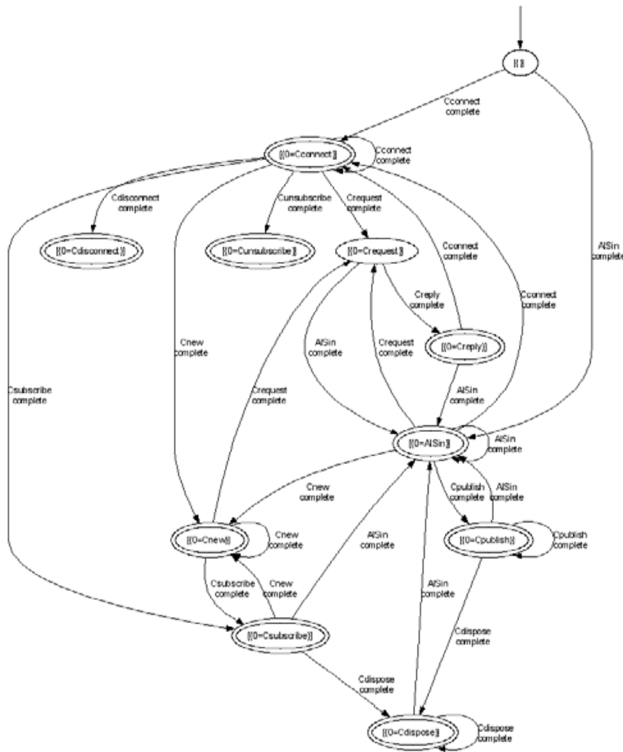


Figure 5. ProM visualization of the final execution profile.

The generated EFSM model includes everything needed by the MBT tool, including the test oracles. These oracles are generated from the ProM state-machine model and from the invariant model produced by the Daikon[3] tool, both generated from the EP trace. The generated test oracles check that the interactions in the tests generated by the MBT tool match those in the state-machine and use the invariants check for the correctness of return values. These need to be checked by the user manually to see that they do not miss any behavioral details, such as self-loops, or creation of complex objects.

Without going into details (due to space limitations of the paper and to retain focus), a basic approach to refine the generated model is to enable the SUT states and transitions one at a time and check their correctness against the specification. With this approach the building of the model also supports the PC process, as the MBT model can be executed and it exactly tells how it matches or differs from the actual implementation. This process is illustrated in Figure 6, Figure 7, and Figure 8. In Figure 6 the first state has been enabled. In Figure 7 two more states have been enabled. In Figure 8 one additional state has been enabled. These visualizations are provided by the MBT tool, which allows a feedback loop similar to using a PC tool. As the model refinement process progresses, all states will be enabled one at a time. At the same time, the executed model and the test oracles also give feedback back to the PC process. An abstracted model may hide important details, and when the MBT tool executes it and reports any errors in matching the model against the implementation, it also makes any assumptions in the model clear and checks them, reporting the results.

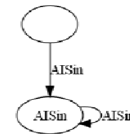


Figure 6. First state enabled in the MBT model.

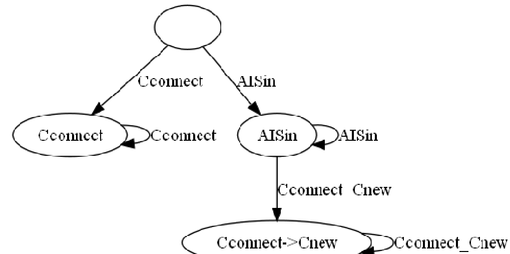


Figure 7. Three states enabled in the MBT model.

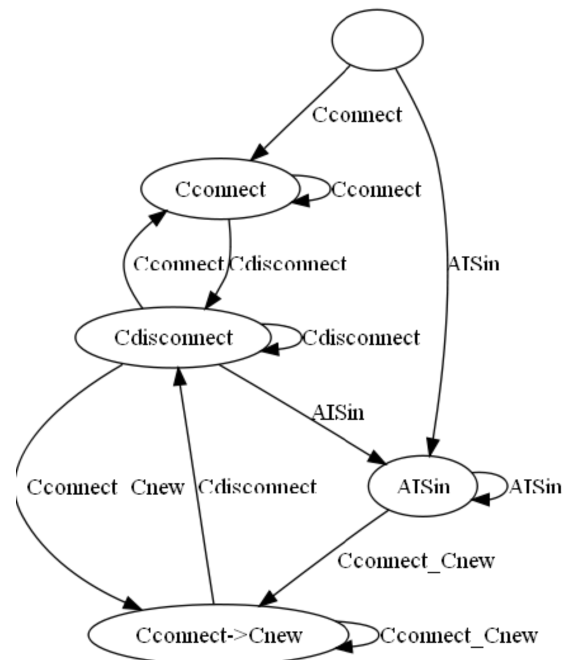


Figure 8. Four states enabled in the MBT model.

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

During this case study, the usefulness of this approach was also demonstrated as it revealed six different, previously unknown, errors in the SUT. These were related to different properties of the SUT, such as incorrect implementation of the specification, missing or ambiguous parts of the specifications, and design errors in implementation details of the SUT.

VI. A Framework and Related Discussion

The existing techniques supporting both program comprehension and test oracle automation presented in section IV and the case example shown in section V share a number of properties. All start with supporting PC in the form of building a model based on the execution profile of the SUT. Test automation is supported by providing means to turn these models into different forms of test oracles for the SUT. From the PC viewpoint, they all apply a hybrid approach that starts with a bottom-up approach of building models from the execution profile, presenting them to the human user for analysis. The human user applies a top-down analysis by using the specification to determine the correctness of the proposed models and to turn them into test oracles.

A. A Framework for User-Assisted Test Oracle Generation

The different representations of the test oracle information used by the different techniques that have been presented also share the property of being invariant representations. Some of these techniques describe the provided oracle information as invariants [14][15][16], however they only discuss invariants as a basis provided by an external tool and not as the underlying concept of the test oracles themselves. However, in practice all oracle information is always a form of an invariant representation of some property of the SUT, which is then verified with the given oracle procedure. In the case of using queries over execution profile traces [11][12], the invariant is that the property expressed by the query holds in all analysed versions of the traces. In the case of the Test Blueprint approach [13], the invariant is that the "side-effects" checked by the created assertions are not changed. The concept of thinking of a test oracle information as a representation of an invariant, and the oracle procedure as an invariant-checker is important as it provides a conceptual framework for creating means to provide fully automated or user-assisted test oracle generation techniques.

In the presented approaches based on dynamic invariants inferred from the SUT execution profile, both the oracle procedure and the oracle information are provided, but the user needs to analyse the provided information and make an assessment if this is correct or not, possibly refining it [14][15][16]. Using queries of the execution profile trace as oracles requires one to define the queries as the oracle information to complete the oracle [11][12]. In the Test Blueprints approach, the user is presented with a visualization of the "side-effects" the execution of a program unit has, which can be used as a basis to write test oracles to verify these "side-effects" [13]. In the MBT model generation approach, both the oracle procedure and oracle information are provided and the user must check the information and refine the generated model as needed. All these approaches require the user to provide the oracle information or to refine it, while the oracle procedure is provided. The invariant- and model-based techniques can be seen as more advanced in their support for the user as they provide (generate) the initial (oracle information) model in a form directly executable as a test oracle, and which can then be analysed and refined by the user.

From these different techniques, it is possible to derive a set of guidelines for what to provide to the user when providing PC related techniques to assist in automated generation of test oracles. The items provide a framework for using program comprehension techniques to provide automated assistance for a user in generating test oracles, and can be summarized as providing the user with:

- An invariant notation suitable for the chosen oracle information.
- (a basis for) The oracle information, i.e. a set of invariants describing a meaningful properties of the SUT.
- The oracle procedure, i.e. an invariant checker.
- (an automated) Means to turn the oracle information into a test case with the oracle procedure.
- Assistance for the user to analyse (comprehend) the generated oracle information.
- Possibility to refine the oracle information.
- Means to (execute the model and) verify the complete oracle, i.e. an automated invariant-checker.

B. Related Discussion

The oracle procedure relates to the oracle information and how it needs to be processed and analysed. In many cases this can be simple, for example verifying that the output from a method call always has a value smaller than 100 (for invariant $x < 100$) can be implemented with a single assert statement. This is closely tied to the test automation platform used, such as a unit testing tool that provides the assertion facility, and the (program comprehension) tools used to provide the model of the execution profile that is used as a basis for the oracle information.

The basis for the oracle information as described in the techniques reviewed in this paper is formed from the invariant- and state-machine models of the SUT created based on the execution profile. Thus they already provide an abstraction generated based on the execution profile that the user can turn into or refine to produce the required oracle information.

Turning the oracle procedure and information into a test case requires mapping the oracle information to the oracle procedure. For example, in the MBT case example, this requires parsing both the invariant and state-machine models and turning them into an EFSM model. In the context of using PC tool, when they provide access to their internal model representations this provides the best support for both PC and test oracle automation as shown in section V.

As any tests generated in this way are based on the execution profile and are thus limited by what executions it contains, the user must be able to analyse the provided test

oracles and to refine the model to match the specification. Since the model describes the actual behaviour of the SUT, the user must also be able to verify that this is actually the expected behaviour of the SUT as expressed by its specification. This highlights the need for means to analyse the models with tools such as those used in PC. This was demonstrated in the MBT case example, where the resulting state-machine could be visualized both from the execution profile and after being generated into an EFSM. In the same case example, it was demonstrated how the model assumptions can be made clear and verified (comparing the generated test oracle vs. the specification) by the PC related visualization and analysis, and by executing the model with the test automation tool.

The process as described by the guidelines matches that of both program comprehension and test oracle automation as described in Figure 3. The user starts with the model of what is the expected behaviour of the SUT, such as its specification. Using the tools provided, the execution profile is captured and turned into a model to be used as a basis for defining the oracle information. The user analyses this information, refines the model and verifies its correctness. As required, both of these models are iteratively refined according to findings in the verification phase.

These examples show that using PC related concepts and techniques can be helpful in the context of providing automated support for test oracle generation. However, as discussed before, and also noted by Cornelissen et al. [21], there is a lack of studies on how the PC techniques support actual humans in their work. This also makes it more difficult to take these approaches and consider how they could be applied in the context of test oracle generation. The example shown in section V uses PC techniques to generate and visualize state-machines as a basis for generating a test model, including a test oracle. Although there are tools for state-machine generation in PC, there are not many empirical studies on their use [21]. Still, in the case study described in this paper, one tool was used to aid in test oracle generation with a straight-forward approach as described in section V. However, more comprehensive studies in application of PC techniques would make the process of applying them for test oracle automation easier. Similarly, this could be eased by providing support for accessing and using the models provided by the PC tools externally from other tools. This support and the framework presented in this section also provide a basis for creating more powerful techniques for user-assisted test oracle generation.

VII. Conclusions and Future Work

In the context of test automation, the creation of test oracles is one of the most difficult parts to automate. This is also visible in the limited number of papers that address the test oracle automation problem. This paper addressed this issue by providing a framework for applying techniques from the field of program comprehension to provide automated assistance to the user in creating test oracles. Related work on this topic was reviewed, analysed, and brought together with the concept of program comprehension for test oracle automation. The concept was illustrated with a practical example of generating models usable as a basis for test oracles in model-based testing. By comparison of this example with related work, a framework was provided for applying program comprehension techniques to provide automated assistance for users in creating test oracles. The provided framework describes test oracles as invariant-checkers and provides a set of guidelines for providing automated assistance to the user in generating these invariant-checkers based on information captured from the program execution with the help of dynamic analysis techniques. The provided framework helps with providing more powerful techniques to assist in the test oracle generation process.

This paper also highlighted need for future work in the field of program comprehension to identify how a human user actually makes use of a program comprehension technique. This information is needed in order to automate the use of these techniques as much as possible in the context of the framework presented here. In the field of test automation, interesting future work includes making use of more program comprehension techniques to support user-assisted test oracle generation.

Acknowledgment

This work has been supported by the Nokia Foundation. The author wishes to thank Eric Verbeek for his help with the ProM tool, and Andy Zaidman and the anonymous reviewers for their helpful comments on improving the paper.

References

[1] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Proc. Future of Software Engineering (FOSE'07)*, 2007.

[2] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," in *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'07)*, Dubrovnic, Croatia, 2007, pp. 185-194.

[3] Ali Mesbah and Arie van Deursen, "Invariant-Based Testing of Ajax User Interfaces," in *Proc. 31st Int'l. Conf. on Software Eng.*, Vancouver, Canada, 2009.

[4] H. Sneed, "Program Comprehension for the Purpose of Testing," in *Proc. 14th Int'l. Workshop on Program Comprehension (IWPC'04)*, 2004.

[5] M-A. Storey, "Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future," *Software Quality Journal*, vol. 14, no. 3, pp. 183-208, Sept. 2006.

[6] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley, "Specification-Based Test Oracles for Reactive Systems," in *Proc. 14th Int'l. Conf. on Software Eng. (ICSE'92)*, Melbourne, Australia, 1992, pp. 105-118.

[7] X. Yaun and A. M. Memon, "Using GUI Run-Time

State as Feedback to Generate Test Cases," in *Proc. 29th Int'l. Conf. on Software Eng. (ICSE'07)*, 2007.

[8] Atif Memon and Qing Xie, "Using Transient/Persistent Errors to Develop Automated Test Oracles for Event-Driven Software," in *Proc. 19th Int'l. Conf. on Automated Software Eng. (ASE'04)*, 2004.

[9] Murani Haran, Alan Karr, Michael Last, Alessandro Orso, Adam A. Porter, Ashish Sanil, and Sandro Fouché, "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks," *IEEE Transactions on Software Eng.*, vol. 33, no. 5, pp. 287-304, May 2007.

[10] J. H. Andrews and Y. Zhang, "General Test Result Checking with Log File Analysis," *IEEE Transaction on Software Eng.*, vol. 29, no. 7, pp. 634-648, July 2003.

[11] S. Ducasse, T. Gîrba, and R. Wuyts, "Object-Oriented Legacy System Trace-Based Logic Testing," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR'06)*, 2006.

[12] C. D. Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt, "An Approach to High-Level Behavioral Program Documentation Allowing Lightweight Verification," in *Proc. 14th Int'l. Conf. on Program Comprehension (ICPC'06)*, 2006.

[13] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz, "Test Blueprints - Exposing Side Effects in Execution Traces to Support Writing Unit Tests," in *Proc. 12th European Conf. on Software Maintenance and Reengineering (CSMR'08)*, 2008, pp. 83-92.

[14] M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," in *Proc. Int'l. Symposium on Software Testing and Analysis (ISSTA'06)*, Portland, Maine, 2006, pp. 169-179.

[15] C. Pacheso and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *Proc. European Conf. on Object-Oriented Programming (ECOOP'05)*, 2005, pp. 504-527.

[16] T. Xie and D. Notkin, "Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions," *Journal of Automated Software Engineering*, vol. 13, no. 3, pp. 345-371, July 2006.

[17] K. Magel, "A Theory of Small Program Complexity," *ACM SIGPLAN Notices*, vol. 17, no. 3, 1982.

[18] A. von Myrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *IEEE Computer*, vol. 28, no. 8, pp. 44-55, August 1995.

[19] N. Pennington, "Stimulus Structures and Mental Represntations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, vol. 19, pp. 295-341, 1987.

[20] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l. Jounral of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[21] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transaction on Software Eng.*, 2009.

[22] C. Bennett, D. Myers, M-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A Survey and Evaluation of Tool Features for Understanding Reverse-Engineered Sequence Diagrams," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 291-315, July 2008.

[23] L. C. Briand, Y. Labiche, and J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Transactions on Software Eng.*, vol. 32, no. 9, pp. 642-663, Sept. 2006.

[24] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Proc. 30th Int'l. Conf. on Software Eng. (ICSE'08)*, Leipzig, Germany, 2008, pp. 501-510.

[25] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe, "Automated Discovery of State Transitions and their Functions in Source Code," *Software Testing, Verification and Reliability*, vol. 18, no. 2, pp. 99-121, June 2008.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Eng.*, vol. 27, no. 2, pp. 99-123, Feb. 2001.

[27] J. E. Cook and Z. Du, "Discovering Thread Interactions in a Concurrent System," *Journal of Systems and Software*, vol. 77, no. 3, pp. 285-297, Sept. 2005.

[28] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering Architectures from Running Systems," *IEEE Transactions on Software Eng.*, vol. 32, no. 7, pp. 454-466, July 2006.

[29] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther, "Procee Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting," *Software and Systems Modeling (SoSyM)*, 2009.

Series title, number and report code of publication

VTT Publications 727
VTT-PUBS-727

Author(s)
Teemu Kanstrén

Title

A Framework for Observation-Based Modelling in Model-Based Testing

Abstract

In the context of software engineering, test automation as a field of research has been around for a very long time. Yet, testing and related concepts are still generally considered to be one of the most time-consuming and expensive parts of the software life cycle. Although it is a field with a relatively long research background, many existing test automation systems are still relatively simple and not very different from the early days. They still focus on executing an existing, usually manually crafted, set of tests over and over again.

One approach that has also been around for a relatively long time but has only recently begun to attract considerable interest in the domain of software testing is model-based testing. In model-based testing, the system under test is represented by a model describing its expected behaviour at a higher abstraction level, and a set of chosen algorithms are used to generate tests from this model. Currently, these models need to be manually crafted from the specification.

This thesis presents an approach for observation-based modelling in model-based testing and aims to provide automated assistance for model creation. This includes design and architectural solutions to support observation and testing of the system, analysis of different types of executions used as a basis for observations, and finally combines the different viewpoints to provide automated tool support to generate an initial test model, based on the captured observations, that is suitable for use in model-based testing. This model is then refined and verified against the specification. As the approach reverses the traditional model-based testing approach of going from specification to implementation, to going from implementation to specification, guidelines for its application are also presented. The research uses a constructive approach, in which a problem is identified, a construct to address the problem is designed and implemented, and finally the results are evaluated.

The approach has been evaluated in the context of a practical system in which its application discovered several previously unknown bugs in the implementation of the system under test. Its effectiveness was also demonstrated by generating a highly complete model and showing how the completed model provides additional test coverage both in terms of code covered and injected faults discovered (test mutants killed).

ISBN
978-951-38-7376-9 (softback ed.)
978-951-38-7377-6 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN	Project number
VTT Publications 1235-0621 (softback ed.) 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	33451

Date	Language	Pages
January 2010	English	93 p. + app. 118 p.

Name of project	Commissioned by

Keywords	Publisher
Model-based testing, test automation, observation-based modelling, test generation	VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4520 Fax +358 20 722 4374

VTT CREATES BUSINESS FROM TECHNOLOGY
Technology and market foresight • Strategic research • Product and service development • IPR and licensing
• Assessments, testing, inspection, certification • Technology and innovation management • Technology partnership

Software testing is one of the most time consuming and expensive parts of the software development lifecycle. Advanced techniques such as model-based testing (MBT) promise to ease this process. However taking these techniques into use can require significant investments and expertise, hindering their success.

This dissertation presents a framework for observation-based modelling (OBM) for MBT. OBM uses a set of observations (trace) captured from actual execution scenarios for an existing system as a basis for an advanced starting point for testing and verification with the help of a MBT tool. The approach helps verifying the correctness of the implementation vs the specification, and testing for errors in the implementation details. Providing advanced starting points for MBT testing techniques and guidelines for their use reduces the cost and effort needed in their adoption and provides for more advanced automated testing and verification support.

ISBN 978-951-38-7376-9 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 978-951-38-7377-6 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)