Model-based Testing in Practice: 2nd Workshop on Model-based Testing in Practice (MoTiP 2009)

June 23, 2009 Enschede The Netherlands

Editors:
Thomas Bauer
Hajo Eichler
Axel Rennoch
Sebastian Wieczorek
Thomas Bauer  
Hajo Eichler  
Axel Rennoch  
Sebastian Wieczorek (Eds.)

Model-based Testing in Practice

2nd Workshop on Model-based Testing in Practice  
(MoTiP 2009)

In Conjunction with the  

5th European Conference on Model-Driven Architecture  
(ECMDA 2009)

Enschede, The Netherlands, June 23, 2009

Proceedings
Preface

This volume contains the proceedings of the 2nd Workshop on Model-based Testing in Practice (MoTiP) held on 23 June 2009 in Enschede, The Netherlands, in conjunction with the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA 2009).

The objective of the MoTiP 2009 workshop is to bring together industry and academia by providing a platform for interaction and collaboration. The continuing industry trend to raise software complexity by increasing the functionality and accessibility of software and electronic components leads to an ever-growing demand for techniques to ensure software quality. At the same time, software companies are shortening development cycles to respond to the customers demand for fast and flexible solutions. In order to remain competitive, early and continuous consideration and assurance of system quality becomes an asset of ever-increasing importance in industrial software development.

Model-based approaches are not only able to provide effective quality assurance, but also help to evaluate and control the coverage, costs, and risks related to testing efforts. Both – the effectiveness and the efficiency of testing – can be handled by model-based approaches within integrated system and test development for software-intensive systems. While the software industry starts to adopt model-based testing techniques on a large scale, promising research ideas are emerging that have the potential to answer many of today’s industrial challenges. Therefore the MoTiP 2009 workshop brings together practitioners and researchers to initiate a dialog much-needed.

The papers in this volume are representative of current industrial and research activities on Model-based Testing. There have been 14 submissions to this workshop, and 10 papers have been accepted for publication and oral presentations. All selected papers are of high quality, thanks to the professionalism of the authors, reviewers, and program committee members.

We would like to take this opportunity to thank the people who have contributed to the MoTiP 2009 workshop. We want to thank all authors and reviewers for their valuable contributions, and we wish them a successful continuation of their work in this area. Finally, we thank the organization of the ECMDA 2009 conference in which this workshop has been embedded.

June 2009

Thomas Bauer
Hajo Eichler
Axel Rennoch
Sebastian Wieczorek
Organisation

Workshop Chairs

Thomas Bauer  Fraunhofer IESE, Germany  
Hajo Eichler  IKV++, Germany  
Axel Rennoch  Fraunhofer FOKUS, Germany  
Sebastian Wieczorek  SAP Research, Germany

Programme Committee

Fevzi Belli  Uni Paderborn, Germany  
Michael Ebert  Robert Bosch GmbH, Germany  
Juhan Ernits  University of Birmingham, UK  
Robert Eschbach  Fraunhofer IESE, Germany  
Andreas Hoffmann  Fraunhofer FOKUS, Germany  
Jacques Kanuga  Daimler AG, Germany  
Andrei Kirshin  IBM Haifa, Israel  
Andres Kull  Elvior, Estonia  
Raluca Lefticaru  Universitatea din Pitesti, Romania  
Bruno Legeard  Smartesting, France  
Jesse Poore  University of Tennessee, USA  
Alexander Pretschner  TU Kaiserslautern, Germany  
Christopher Robinson-Mallett  Berner & Mattner, Germany  
Ina Schieferdecker  TU Berlin, Germany  
Alin Stefanescu  SAP Research, Germany  
Detlef Streiterdt  ABB Corporate Research Center, Germany  
Dragos Truscan  Abo Akademi University, Finland  
Colin Willcock  Nokia Siemens Networks, Germany
# Table of Contents

**Session 1: MBT for Embedded Systems**

Combining Combinatorial and Model-based Test Approaches for Highly Configurable Safety-critical Systems ........................................... 9

*Thomas Bauer, Robert Eschbach, Martin Größl, Tanvir Hussain, Detlef Streitferdt and Florian Kantz*

An Approach for Test Derivation from System Architecture Models applied to Embedded Systems ................................................ 23

*George Din and Klaus-Dietrich Engel*

Model Based Statistical Testing and Time ............................................................... 33

*Frank Böhr*

Test Case Generation for Embedded Automotive Systems: A Semantics Preserving Model Transformation ....................................................... 43

*Chimistiu Valentin, Pelschl Berhard and Schwarzl Christian*

**Session 2: UML-based MBT Approaches**

Transformational Support for Model-Based Testing - from UML to QML ............. 55

*Fredrik Abbors, Tuomas Pääjärvi, Risto Teittinen, Dragos Truscan and Johan Lilius*

Combining Sequences and State Machines to Build Complex Test Cases........... 65

*Dehla Sokenou and Stephan Weissleder*

Model Based Testing of Web Service Composition Using UML Profile ................. 75

*Muhammad Zohaib Z. Iqbal and Zafar I. Malik*

**Session 3: Tool Support for MBT**

Case Study Based Performance Evaluation of Reactive Planning Tester ............... 87

*Fredrik Andres Kull, Kullo Raiend, Jüri Vain and Marko Küdramées*

Compliance Test Framework .................................................................................... 97

*Leon Bounmeester, Guy Broadfoot and Philippa Hopcroft*

Emerging Patterns for Testing Model Management Tools .................................... 107

*Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Tara Gilliam and Fiona A.C. Polack*
SESSION 1
MBT for Embedded Systems
Combining combinatorial and model-based test approaches for highly configurable safety-critical systems

Thomas Bauer*, Robert Eschbach*, Martin Größl*, Tanvir Hussain*, Detlef Streitferdt*, Florian Kantz*

*Fraunhofer IESE
Kaiserslautern, Germany
{bauer|eschbach|groessl|hussain}@iese.fhg.de

*ABB Corporate Research
Ladenburg, Germany
{detlef.streitferdt | florian.katz}@de.abb.com

Abstract. This paper describes a systematic model-based test approach for parameterized systems with a large configuration space called REDUCE. The major goal is to reduce the complexity of the test problem represented by the large number of configurations and possible test cases. The REDUCE approach presented here combines combinatorial and model-based techniques to determine test configurations and to generate automatically meaningful test cases. Combinatorics is used to restrict the number of system configurations and to select a small and valid set of test configurations. Model-based techniques are applied to build a test model that describes the relevant stimulation sequences of the test object. Test models enable the automated generation of representative test cases and their automated evaluation. For each selected configuration, a specific test model can be derived. REDUCE has been developed and applied to a softstarter, an industrial application study from the automation domain.

Keywords: model-based testing, statistical testing, combinatorial testing, n-wise parameter coverage

1 Introduction

Safety-critical embedded systems such as control applications in industrial automation are complex due to their broad range of possible applications. To cover this range, typical embedded systems come with a huge set of configuration parameters. Each of these parameters has an influence on the overall system behavior.

From the perspective of quality assurance it is not clear how to test such a huge configuration space w.r.t. different quality attributes such as safety or reliability. Reliability is one of the essential non-functional quality properties of software systems required by contract or standards like the IEC 61508 [3] or IEC 61511 [4]. Metrics as mean time between failures (MTBF) or failure rate are typical reliability indicators.

It is obvious that exhaustive testing is not realistic, even for simpler systems. The questions are which test sample should be drawn from the input domain, which configuration should be chosen, and which configurations need not be considered in order to make realistic reliability estimations for such systems.
In this paper, we present REDUCE, an approach that has been developed and evaluated in the research projects D-Mint and DNT. REDUCE consists of several activities that help to significantly reduce the complexity of the test problem until reliability estimations based on testing become feasible. REDUCE combines several classic testing techniques, such as equivalence partitioning, boundary analysis, and n-wise parameter coverage, in a completely new way. For automated test case generation a model-based test approach (MBT) is used. A generic test model is built from the system specification, which allows the generation of valid configuration-specific test models and test cases.

The approach presented here has been successfully applied to a representative safety-critical system from the industrial automation domain, a softstarter from ABB (see section 3). The first results of the initial application are shown in this paper.

The paper is structured as follows: Section 2 shows the related work. The test object and the challenges of the industrial automation domain are described in section 3. The REDUCE approach is presented in section 4. The basics of the combinatorial and statistical approaches of REDUCE are described in sections 5 and 6. Section 7 covers the approach-specific test models. The test case generation, execution, and evaluation are covered in sections 8, 9, and 10. The paper concludes with the validation and the summary in the sections 11 and 12.

2 Related work

Various works deal with testing of highly configurable safety-critical embedded systems. The focus of research can be classified into two sections: testing of software product lines and combinatorial testing that selects configuration parameter.

Most of the research work is done in the field of product line engineering. The main aspect in [2] is the reusability of test cases for different configurations. McGregor [8] focuses on specific points of the variations between products that are the basis for the composition of test cases. Abstract test scenarios are modeled as use cases representing the product requirements model. These scenarios are parameterized with characteristic values to instantiate the different products. Both approaches do not use model-based testing techniques. Such a technique is introduced in [5] and [10]. The publications describe how to represent use cases with UML extensions and consider the variability of software product lines. This method supports derivation of test cases from these models.

Another MBT approach [12] provides the derivation of application-specific test scenarios from use cases and activity charts on the domain level, which contain the variations of the different products.

The second area of interest is the combinatorial way. In this field research is performed in experiment design [2]. The challenge is to find an optimal set of configurations that satisfies maximal coverage criteria for each test case. Results of failure interactions have been published in an empirical study [6]. The results showed a dependency of two to six parameters in the case studies.

1 D-Mint project web-site: www.d-mint.org
2 DNT project web-site: www.nutzfahrzeugcluster.fraunhofer.de
3 Example application from the industrial automation domain

For the automating processes in a plant such as the processing of raw oil to gasoline, or airport luggage conveyor belts, electric motors are used. The operation of such motors is mostly computer-controlled and thus, the motors are operated remotely. Here, we consider the following simple operational scenarios:

- **Turn a motor on/off.** For small motors and cases where no motor regulation is needed such as small fans.
- **Smoothly ramp up/down a motor.** For large motors where the peak energy drawn from the power network would cause unwanted disturbances and for cases like conveyor belts, where we do not want the transported goods to drop if the belt starts too rapidly. Or consider an elevator, where we do not accept an immediate start/stop of the cabin.
- **Vary the speed of the motor.** For cases where continuously controlled speed changes are needed such as robot arms, which move at different speeds.

In both scenarios, we use embedded systems – motor starters or softstarters, which are the example of a safety-critical embedded system discussed in this paper – to operate and monitor the motor. Monitoring of the motor will detect, e.g. a locked motor, an overheated motor, or disturbances in the power supply network that might damage the motor.

Although the functionality of a softstarter looks fairly simple (see the scenarios above), the configuration of such a device is quite complex. Besides the start/stop functionality that must be adopted to the respective application, the softstarter also contains fault, warning, and protection handling and the possibility to control some special sensors and actuators.

The configuration will be done by setting the values of 130 parameters. This complexity forms a challenge in respect to testing the softstarter. Even if the execution of a single test case took only one minute, this number of configurations could not be tested. The only way to handle this complexity is the structured selection of a subset of parameter configurations.

For future development efforts, we foresee the need for improved testing methods and tools. To address these challenges, we state the following requirements:

- Automatically derive system test cases according to the system usage.
- Automatically derive parameter sets to be used with each test case.
- The selection process and the criteria for test cases as well as parameter sets should be transparent to certification authorities (according to IEC 61508).
- The execution of the automatically derived test cases should be automated as well.

4 The overall approach: REDUCE

REDUCE is capable of treating huge configuration spaces and delivering meaningful test results based on different representative operational profiles of the test object that can be used for reliability estimations. Starting from the system requirements we divide the test problem into (i) the determination of possible system configurations and (ii) the generation of meaningful test cases.
The REDUCE approach is presented in Figure 1. Based on the system configuration, the system functionalities are configured. Different kinds of functionalities exist. Control functions are used for the normal operation of the device, while protection functions reduce risks associated with attached physical and logical devices and warning functions support the usage by staff (for example workers) integrated into specific workflows. A possible extension of the approach would also check the system behavior after hardware failures (sensors, actuators).

Figure 2 shows the steps and artifacts of the proposed test approach. In two parallel steps, the system configurations are determined and the generic test model contains the possible system usages and expected system responses considering the high configurability of the test object.

A system configuration is a valid set of parameters and values. The validity of a parameter set is determined by constraints that are derived from requirements and expert knowledge. A large number of valid system configurations exist as result of the number of parameters and possible parameter values.

In a parallel step, a generic test model is built, describing all relevant system inputs and usages from the tester's and user's point of view. Therefore, test models are also called usage models. The generic test model is derived from requirements and represents the formalized functional requirements with importance annotations.
Later, concrete test models are derived from the abstract test model. In general, a concrete test model is a subset of the generic model considering the particularities of each configuration. From the concrete test models, test cases are generated that are executed on the test object. The test results are recorded and analyzed in order to determine the current quality of the test object.

5 Model-based statistical testing

Model-based statistical testing (MBST) is a black-box testing technique that enables the generation of representative tests from the tester’s or user’s perspective ([16], [11]). MBST has been extended for risk-based testing ([13], [15], [17]) and applied to safety-critical embedded systems ([14], [18]). In the REDUCE approach, we use MBST to construct the generic test models and to automatically generate test cases from the concrete test models.

Figure 3 shows the steps of the model-based statistical testing approach. In the first stage, a test model is built from the requirements that represents relevant system inputs and usages and the expected system responses. The test automation stage deals with the automated generation, execution, and evaluation of test.

In the test modeling stage the original system requirements are systematically inspected by the sequence-based specification method to determine the system boundary and the stimuli and responses of the test object. The next step is the sequence enumeration which aims at systematically writing down each possible stimulus sequence, starting with sequences of length one. A sequence of stimuli represents one history of the usage of the system under test. Additionally, every sequence is compared to previously analyzed sequences w.r.t. equivalence. Two sequences are equivalent if their responses to future stimuli are identical. Illegal and equivalent sequences are not extended in the ongoing sequence enumeration. The enumeration stops if all sequences are illegal or reduced to equivalent sequences.

Certain inconsistencies, missing specifications, unclear requirements, and vague formulations can be found in the requirements document during sequence enumeration. All construction decisions for equivalence, system responses, and requirements coverage are traceable to requirements by following requirements traces attached to the enumerations.
The list of input-output sequences will be mapped to a Mealy machine. The Mealy machine represents the structure of the usage model and describes all possible usages identified from the system requirements. By adding probabilities to the model transitions, the state machine becomes a discrete-time Markov chain (DTMC). The probabilities reflect the expected usage or criticality of system inputs.

Test models have particular states for the initialization (START) and finalization (EXIT) of test cases. A test case is an arbitrary path through the model from START to EXIT. The state START describes the system state at the beginning of a test case. The state EXIT marks the end of a test case and can be reached from all states where a test case can end.

In the test automation stage test cases are generated as paths through the test model from the START to the EXIT state. Different strategies for automated test case generation exist, e.g.:

• **Model coverage tests** make sure that the whole model is covered by test cases. This means that each transition and each state of the test model is tested.

• **Random tests** are randomly generated paths through the test model based on a usage profile, e.g., frequency or criticality.

The transitions of the test model are annotated with scripts for the test runner. Hence, during the generation of a specific test case, the scripts of the transitions lying on the path will be aggregated one after another to build a concrete test case that is executable in the selected test environment.

During test execution, the number of executions with and without failure is counted for each transition. Based on the usage profile and the failure statistics, the reliability of the test object is estimated after each test run. Reliability in MBST is the probability of a no-failure operation [9]. In the case of a criticality profile, safety compliance is estimated instead of reliability.

### 6 Combinatorics and configurations

A particular configuration influences the behavior of the software, which is mostly visible through its response to sensor signals, activation of output signals, etc. We identified 61 relevant parameters, classified into three groups depending on its value type: numeric, Boolean and enumerative. Parameters having numeric values can be partitioned into equivalent classes such that constituents of a class do not differ in terms of their effect on the behavior of the software. Furthermore, certain parameters are dependent on each other in such a way that each assignment of one restrains the assignment of the rest. Parameters can further be divided into 5 groups depending on their effects on the type of functionalities (cf. Figure 1) that a softstarter should provide.

<table>
<thead>
<tr>
<th>Affected functionality</th>
<th>No. of Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Numeric</td>
</tr>
<tr>
<td>General</td>
<td>6</td>
</tr>
<tr>
<td>Protection</td>
<td>5</td>
</tr>
<tr>
<td>Fault</td>
<td>0</td>
</tr>
<tr>
<td>Warning</td>
<td>3</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>13</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>
Table 1 gives an overview of the above-mentioned classification of the parameters. The user interface is assumed to be beyond the boundary of the test object and delivers only valid configurations which satisfy the restrictions imposed due to the dependencies of the parameters. But even after considering these dependencies the possible number of representative configurations is higher than $10^{30}$. To reduce the number to a feasible one, only the effects of pairwise interactions among the parameters are observed (see [7]).

For a large number of parameters, as in the current case, pairwise combinations provide a tremendous advantage for reducing the total number of configurations. Most of the direct dependencies (i.e., one value assignment constraining the other, as mentioned above) among the parameters are pairwise.

7 Test models for configurations

Different system configurations require different test models. In the REDUCE approach, we use a generic test model that allows the generation of configuration-specific test models. The generic test model is derived from the system specification [1] by applying the approach from section 5.

Table 2: SBS stimuli table

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault_TSC_STOP-M</td>
<td>Thyristor short circuit fault</td>
<td>(manual start)</td>
</tr>
<tr>
<td>Fault_TSC_STOP-A</td>
<td>Thyristor short circuit fault</td>
<td>(automatic start)</td>
</tr>
<tr>
<td>Fault_TSC_IND</td>
<td>Thyristor short circuit fault</td>
<td>(indication)</td>
</tr>
<tr>
<td>Hi_CurrProt</td>
<td>High Current Protection $I &gt; 8*I_e$</td>
<td></td>
</tr>
<tr>
<td>!Hi_CurrProt</td>
<td>High Current Protection gone</td>
<td>$I &gt; 8*I_e$ (time &lt; 200ms)</td>
</tr>
<tr>
<td>Hi_Time</td>
<td>Time for High Current Protection</td>
<td>Time greater than 200ms</td>
</tr>
</tbody>
</table>

The model describes the configuration-independent functional aspects of the device, i.e. the mapping from stimuli sequences to responses. An extraction of the system stimuli is presented in Table 2. These parameters enable a sequence enumeration process (see section 5 for details). A subset of the sequence enumeration is described in Table 3. The sequence enumeration describes prefix sequences, stimuli, expected response and equivalences.

Table 3: SBS sequence enumeration

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Stimulus</th>
<th>Equivalent sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Fault_TSC_STOP-M</td>
<td>Stopped</td>
</tr>
<tr>
<td>Running</td>
<td>Fault_TSC_STOP-A</td>
<td>Start</td>
</tr>
<tr>
<td>Running</td>
<td>Fault_TSC_IND</td>
<td>Running</td>
</tr>
<tr>
<td>Run.HiCurr Prot</td>
<td>Hi_Time</td>
<td>Stopped</td>
</tr>
</tbody>
</table>

In our case study the sequence enumeration process results in a generic model which contains several transitions for a parameter e.g. Fault_TSC (manual / automatic / indication). At the current stage of the generic test model we have used the uniform probability distribution for all transitions. In the future, we will consider the risk analysis of the product to enable risk-based testing focusing on critical functions.

In order to derive valid test cases, a configuration-specific subset of the generic model has to be extracted. In the configuration-specific test model, irrelevant conditional
transitions are removed and concrete parameters are set. This implies the mapping of parameter values to specific transitions. As a result of the reduction step concrete models are generated. The relation of generic and concrete test models is used for the test evaluation in section 10.

Figure 4: Generic and configuration-specific test models

An example of such a model reduction is presented in Figure 4. The generic test model has three transitions for a particular failure case. The concrete model has exactly one transition per case.

8 Test case generation

Test cases are configuration-specific, i.e. the paths traversed by the test case and the test oracle may differ. From every configuration-specific test model, a set of test cases was generated comprising model coverage set and random tests to get meaningful test results.

The number of test cases to be generated is linked to the planned test effort. Taking agile development as an example, the short cycles for a working system (e.g. a work week of five days) would result in a testing period of about 50 hours between Friday evening and Monday morning, when the next cycle starts. Given an average of one minute per executed test case, this leads to about 3000 test cases that can be executed over the weekend. The approach described in this paper enables the efficient adaptation of the test model and the parameter set with the generation of test cases to meet the testing requirements.

9 Test execution

Each transition in the test model is annotated with a test script. The scripts have to conform to a test interface defined for the test execution environment. This test interface holds all functions to manipulate the softstarter itself as well as the test environment. The following list briefly describes the test interface functions.

In Figure 5, the interface all scripts need to conform to is made up of \texttt{sut\_operate} and \texttt{env\_operate}. The \texttt{test\_environment} embeds the softstarter electrically. It connects the softstarter to the motor in different ways. It will also connect additional test hardware as needed, e.g., a device to increase the current flow through the softstarter to test current protection functionality.
The first part, \textit{sut\_operate}, offers functions for setting and reading parameters stored in the softstarter and operating the keypad. The second part, \textit{env\_operate}, offers functions for connecting the softstarter as desired to the power and the motor, as well as for setting up any additional test hardware. The softstarter is able to measure the motor temperature, which can also be simulated with the test environment.

A more complex topic in the simulation of the environment is the simulation of a load for the softstarter. In the system at hand, this is done by connecting a second motor to the one controlled by the softstarter, which will be operated in generator mode. Using a frequency converter to control the motor in generator mode, it is possible to simulate adjustable load depending the system configuration.

Both interfaces, \textit{sut\_operate} and \textit{env\_operate}, also offer sensor value reading functions. While any test case is running e.g., voltage, current of all three phases, motor rpms, torque, keypad led status, keypad display content can be tracked and analyzed to judge whether the test case failed or not.

\section{Test evaluation}

Test evaluation consists of three-fold activities:

1. \textbf{Evaluation of observable reactions of the software at each test step}: Each test case consists of a number of test steps, each of which stimulates the software under test through a definite trajectory. Each of these trajectories can be mapped to a definite transition on the concrete test model (cf. section 14). Certain test steps expect that the software in reaction manifests itself through observable responses, which might be discrete quantities, e.g., relay output for locked rotor, as well as continuous quantities, e.g., start/stop ramp voltage variation. In other test steps, there is no observable manifestation irrespective of whether the related transition leads to a change of state or not. When the response is of discrete value, it is simple to specify in test oracle and to ascertain the pass/fail verdict. For specifying test oracles for continuous quantities, the test oracle specifies a reference shape. Evaluation is based upon ascertaining whether the actual value

- remained within a tolerable limit beyond the reference shape and/or
- assumed a variation over time that does not cross a defined limit.
2. **Evaluation of each test case:** A test case is composed of a number of the above-mentioned steps referring to a traversal starting from the initial state of the concrete test model to the unique exit state. The appearance of a failure at a certain step may lead to one of the two consequences:

- stopping of the test case execution or
- marking the step as failed and continuing traversal till the exit state.

3. **Evaluation in terms of coverage criteria:** This activity aims at reducing the number of test cases that have to be executed for each system configuration. The generic test model \( g \) along with the concrete test models \( (c_1, c_2, ..., c_n) \) can be related in terms of an algebraic structure called semi-lattice. A semi-lattice is a partially ordered set (poset) closed under one of two binary operations, either supremum (join) or infimum (meet). The test models constitute a set \( S \), which is closed under meet \( (\land) \). This operator can be interpreted as a set theoretic union operator \( (\lor) \) between the sets of transitions of the operand test-models. The generic model \( g \) appears as the unity element, since for any \( c_i \in S, g \land c_i = c_i \). The algebraic structure renders the advantage that for any \( c_k \) that satisfies \( g \land c_k = c_k \), if \( c_k \) is completely covered, a partial coverage of \( c_i \) is also achieved. Only certain additional test cases need to be executed to achieve complete coverage for \( c_i \). Formation of the algebraic structure allows executing tests incrementally, starting with test models lower in the structure and moving towards the top (closer to \( g \)). This saves a huge number of redundant test step traversals.

11 **Results from initial application**

Highly configurable software systems pose an immense challenge as far as testing is concerned. From the testing perspective, the software might be of manageable complexity when a single configuration is considered. Consequently, suitable coverage criteria may be determined either to decide upon termination of testing exercises or to state the confidence achieved regarding the integrity of the software. But the huge constituents of the configuration space multiply the state space to an innumerable one. It is then impossible to choose a suitable coverage criterion that relates to the global state space. Rather, a coverage criterion relating to certain subsets of the complete state space appears to be practicable as well as sufficient in certain cases.

Another important aspect to consider is that coverage criteria as well as the statements relating to these are certainly not objective, as those emphasize certain aspects of the related software, i.e., decision/branch coverage emphasizes the role of branching statements, while statement coverage puts equal emphasis on a branching and a non-branching statement. Therefore, a prudent choice of coverage criteria and their use in the testing process is quite important.

The challenge of the current work was dealt with a divide-and-conquer approach. The division is imposed on the state space of the configurations. The basis of the division is threefold, namely:

1. **Equivalence partitioning and boundary analysis applied to numeric parameters:** Each of the 27 numeric parameters mentioned in Table 1 can assume more than 100 possible values. By applying equivalence partitioning the domain of each was reduced to at most five classes. Boundary and representative values from each class
were chosen to reduce the domain size of each of the numeric parameters by more than 85%.

2. Limiting the combinatorial exploration to only pairs: This certainly needs strong arguments in favor of it, especially related to the case at hand. As far as the test object, i.e., the ABB softstarter, is concerned, pairwise coverage of parameters proves sufficient as explicated in section 4. There are only 1830 pairs that are to be covered on the whole, which can be achieved through 43 configurations, i.e., value assignments. The prudent decision of pairwise coverage shrank the number of configurations from the order of $10^{34}$ to a mere 43.

3. Dividing parameters into disjunctive subsets: Finally, parameters are divided into disjunctive subsets in accordance with their effects on various functionalities of the softstarter. Constituents of each subset are unrelated to the constituents of another subset. Therefore, pairwise coverage of parameters needs only to be applicable within each subset. This reduces the number of required pairs to be considered from 1830 to 439, a reduction by 76%. The subsets, being unrelated, render practicing of a larger degree of combinatorial coverage within each subset still feasible.

The reduced set of configurations is used to derive the concrete test models from an abstract test model. The imposition of an order in terms of the algebraic structure defined in section 10, made it possible to test incrementally through the order. This further ensures the coverage of the test models and the avoidance of the execution of redundant test cases. On the one hand, this results in savings in terms of time and effort involved in test execution. On the other hand, this makes it possible to provide a plausible coverage statement relating to the test models.

12 Conclusions and outlook

We have presented a new approach for testing highly configurable systems combining combinatorial and model-based technologies. A method for generating a small set of valid system configurations was developed by using equivalence partitioning, boundary analysis, and n-wise parameter coverage. For each system configuration, a specific test model was created based on the original system requirements. By applying the semi-lattice approach, we were able to reuse the similarities between different configuration-specific test models to reduce the number of test cases to be generated.

The REDUCE approach was initially applied to an existing application from the industrial automation domain. We were able to significantly reduce the complexity of the test problem of the case study caused by the large configuration space.

A prototypical tool chain that supports the different steps of REDUCE is being developed. In future research, we will focus on the detailed validation and scalability analysis of our approach using the same industrial application. The goal is to compare the existing test approach and REDUCE in terms of effort and efficiency.
13 References


Figure 9: config.-specific test model

Figure 10: generic test model
An Approach for Test Derivation from System Architecture Models applied to Embedded Systems

George Din, Klaus-Dietrich Engel, Axel Rennoch
Fraunhofer Fokus, Kaiserin-Augusta-Allee 31, 10589, Germany
{George.Din, Klaus-Dietrich.Engel, Axel.Rennoch}@fokus.fraunhofer.de

Abstract. The integration of different system aspects and different system models becomes increasingly important for systems that are developed partially in software and partially in hardware (e.g., Hardware in the Loop (HiL) systems used in automotive area). However, the model based testing techniques applied to such systems suffer a lot due to the increase of the complexity of these system models. This paper presents an approach for Architecture Driven Testing (ADT), a technique to derive tests out of, so called, architecture viewpoints. An architecture viewpoint is a simplified representation of the system model on a specific perspective and regards the structure of the system under this perspective. The architecture viewpoints concentrate on specific aspects and allow the combination of the aspects, relations, and models of the various kinds of system components and thereby provide a unified solution.

Keywords: Architecture, Embedded Systems Testing

1 Introduction

Dealing with the complexity of a system is not only a problem in the area of system design but also a challenge when deriving tests for them. In order to handle this complexity, an early consideration of the test design already in the system modeling phase becomes essential in industrial software development. Model-based testing approaches help to efficiently construct tests but also help to evaluate and control the coverage, costs, and risks related to the testing efforts. The effectiveness and the efficiency of testing can be handled by model-based approaches within integrated system and test development. However, current model-based testing approaches consider selected aspects of system models in isolation, e.g. structural or behavioral models.

Using different levels of abstraction and concentrating on specific aspects of the system from different viewpoints[1] proved to be helpful in various situations. This is not only used for software design but also for the construction of buildings. Within this article we present how the abstraction level of system architecture and the use of different viewpoints and the combination of both approaches can be used to derive tests.
The rest of the paper is structured as follows. Section 2 introduces the concept of modelling a system as a set of architecture viewpoints. This concept is used in Section 3 to explain in detail which aspects can be addressed for testing purpose and which kind of tests can be derived out of architecture views. The test derivation method has been used on a case study on car mirror controller which is briefly presented in Section 4. The paper ends with the conclusions.

2 System Architecture and Architecture Viewpoints

On the abstraction level of “System Architecture” a system is described using blocks building up the system, their interconnection through ports and the communication between the blocks down to the communication type and information exchanged between those blocks. The granularity is often determined by using blocks that have a character of some “units of resell or common reuse” that will represent functionality within the system that can also be reused in other contexts in and outside of this system. With this level of abstraction we have a level of granularity that does not describe too many details of the realization of the system on the one hand, but describes enough details on the other hand to see how the blocks will work together to provide the required functionality and features from the requirements specification of the system.

The architectural diagram presented in Figure 1 illustrates a possible design of a speed control system by using diverse functional blocks. For example, the gas pedal is a separate functional block which sends a signal representing the angle of the pedal. That signal is received by another function block synthesizing the angle value to another signal which is sent to the motor in order to increase/decrease the speed. In the same way the functionality of the system can be split into various blocks, each block realizing only a specific function.

![Figure 1. A simple architectural diagram example](image)

Combining the abstraction level of “Architecture Description” of the system and the method of decreasing complexity by applying viewpoints, we get to the concept of describing the system under “Architectural Viewpoints”. This concept has been used in the system design phase, e.g., in the automotive area within the tool PREvision[2] developed by Daimler and Aquintos. Their use of architecture and viewpoints is shown in Figure 2.
Looking at the system from different points of view is a way to deal with complexity. We could for example look at our system from different viewpoints. A viewpoint is the logical view that shows the system in the form of functional blocks to realize the required functionality without taking into account any technical aspects of their realization: e.g. whether they will be realized in soft- or hardware, the hardware they are to be running on as single blocks or in combination with others, the technical resources being available and used in competition with other blocks, etc.. Those technical aspects will be taken into account under another viewpoint, technical view. Here we exactly look at them and assign the functional blocks from the logical view to a realization in or on specific hardware. Another view, the topological view, may pay respect to the aspects where those hardware blocks are located in the system, the wire harnesses and other aspects concerned with the “topology” of the system. Other views may be used and applied as well to get a less complex view on the system by concentrating on a specific aspect of it.

Nevertheless, we need the functional requirements to be linked directly or indirectly to the system architecture artefacts under the different views. The functionality is given in a simple form of Preconditions, Events and Reactions (PER). PERs statements contain references to names of blocks or ports where certain signals are produced or observed. A Precondition or Event is to be linked to the Out-Port of
the functional block that initially sends that signal to the system. A Reaction is to be linked to the In-Port of the functional block where the reaction is to be finally observed. All functional blocks participating in the realization of a specific functional requirement are to be linked to the function block.

Very important in this approach is also to keep track of relations between elements used within the different views. Within the system design as well as testing phase it is of relevance to know which requirement is realized by which component under which viewpoint. Within our example of PREEvision we could have links that specify:

- which functional requirement is realized by the combined action of which blocks in the logical architectural view,
- which blocks from the logical view are realized or executed on which technical blocks in the technical view and
- which technical blocks are located where and where the communication goes through.

3 Architecture viewpoint based test derivation

3.1 Viewpoints information used for testing

Looking at the architectural descriptions on the different layers and the relationships specified, we can identify interesting aspects for testing. Each viewpoint can be used to derive tests.

- **Functional View.** The functional requirements defined in the functional view describe how the system should react for certain stimuli. They can be used directly to derive functional tests which reproduce the stimuli defined in the requirements and correspondingly validate whether the system produces the expected reactions.

- **Logical View.** The realization of multiple functional requirements by the combination of the same sets of function blocks in the logical architecture may cause interferences between the functional blocks and, therefore, cause system malfunction. In this situation, we can use the logical view to detect which particular functional blocks may interfere (e.g., are directly interconnected). Based on this structural information we generate tests to validate whether other functional blocks are affected by functional blocks implementing a certain functional requirement.

- **Technical View.** The mapping of logical function blocks on the same technical blocks in the technical architecture view causes competition for limited hardware resources and, therefore, interference in the execution of those functions which may also cause malfunction of the system. Similar statements hold for the combined use of communication channels. Based on the linking information about which functional blocks are realized by which technical blocks in the technical view, we generate tests for particular technical blocks which might get overloaded during system runtime.
• **Topological View.** The collocation of hardware or wires in the topological view may cause electromagnetic interferences, which may be a reason for the malfunction of the system again.

These aspects of possible failures of the system may drive us to derive specific tests to discover them by testing the system with respect to the specific viewpoint. It is also important to mention, that those aspects might not have been discovered using the requirements specification only.

### 3.2 Types of Tests

In order to derive tests, the information about the logical and technical architecture is combined with the information contained in the functional requirements (in form of PERs). The test derivation strategy is based on two steps: a) decide which functional block (or blocks) should be tested by inspecting the possibility to interfere with other blocks or the possibility to overload, and b) establish which PERs can be applied to the selected functional block in order to validate the functional block in relation with those PERs. We structure these tests into three categories.

**A. Functional tests based on function block diagrams and functional requirements.** These tests address functional blocks in relation with the PERs they need to realize. Assuming that a functional block uses only a sub-set of ports to realize a PER, it is important to check this functionality while the rest of unused ports or functional blocks are stimulated or observed. This method can be applied to logical view, technical view and topological view in order to evaluate if the functional blocks of those views implement correctly the functional requirements.

Examples of such tests are:

1. Validation of PERs when additional ports are stimulated with valid/invalid values. A functional block is tested against a PER while its additional ports (not regarded by the PER) are simulated too. It is assumed that the functional block should satisfy the functional requirement regardless what happens with the additional ports.

2. Validation of additional assumed observations. These tests make additional assumptions (while testing a PER) which need to be observed also as reactions. The additional observations are generated based on patterns. For example, the functional blocks with names which are in antithesis (right, left) cannot react at the same time. (Example: if the control of the mirror is set on left then no reaction should be observed on the right side).

3. Monitoring of function blocks which should not have activity when their neighbors are active. These tests simply listen to additional blocks connected to blocks realizing a PER. The additional blocks are not involved in the realization of the PER, therefore, it is assumed that they should not produce any reaction.
B. Functional tests with sequences of additional events (from type specification).
The definition of a function block in an architecture view contains the data types and all possible values of each in or out port. Another category of tests targets the creation of various sequences of events applied at the input ports while the reactions are validated against the functional requirements. This algorithm can be applied to all functional blocks in the logical view.

C. Load test. Load tests are derived in order to test overload conditions of function blocks at the technical view which realize multiple functional blocks from logical view. This type of tests combines an arbitrary number of test behaviors derived from PERs. The stimuli are applied simultaneously. However, not any combination is possible. The preconditions, events and additional events (if the case) of one test behavior shall not collide with any of the preconditions, events and additional events of the other test behaviors. A collision may occur when two tests regard the same signal but require stimulating different events.

4 Case study: Exterior Mirror

The approach of applying the system architecture abstraction level and viewpoints to decrease complexity and concentrate on specific aspects, step by step, has been applied within a case study from the automotive area. A system architecture (sub) model describing aspects of an exterior mirror, created with PREEvision, has been used as a start point to derive tests. This system is presented Figure 3. From a logical viewpoint the system consists of a mirror side selector (indicating which mirror to be controlled), a mirror position setter (for adjusting the position of the mirror on the selected side), the key status, the mirror controller (which can consist of further functional blocks), and four blocks for controlling the vertical and horizontal move of the mirror on both sides of driver and co-driver.

In Listing 1 we selected a simple functional requirement which is used as example to explain the testing elements which can be derived from the architecture view. The requirement is implemented by the functional blocks presented in Figure 3. The precondition, the events and the reaction apply to the function blocks contained by their description and describe a certain behavior which those blocks should realize.

<table>
<thead>
<tr>
<th>Functional Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition:</strong> Key Status has the setting “KL 15”</td>
</tr>
<tr>
<td><strong>Event:</strong> Mirror Side Selector is set on “Driver”</td>
</tr>
<tr>
<td><strong>Event:</strong> Mirror Position Setter is pressed on “right”</td>
</tr>
<tr>
<td><strong>Reaction:</strong> Driver Mirror Horizontal Control indicates a movement to “right”</td>
</tr>
</tbody>
</table>

Listing 1. Example of functional requirement
Within the case study we applied the test types shown in Section 3.2. For each test derivation algorithm we defined a set of PREEmision model query rules and used the report writing facility to derive the test configuration description, types and templates and test flow from the information in the PREEmision system architecture model which also includes the requirements as test targets.

The tests at the moment are non-executable, only in a textual description. Nevertheless, TTCN-3 [3] textual format could alternatively be generated.

4.1 Test Configurations, Test Data and Test Behavior

The tests are described by means of test configurations, test data and test behavior. We exemplify these elements on a testcase derived by using the algorithm (A.2.) presented in Section 3.2. The testcase is derived by using the functional requirement presented in Listing 1 and the logical view architecture presented in Figure 3. A test configuration identifies the tested functional block (or blocks) named system under test (SUT) and the functional blocks simulated by the test system. The test configuration used in the testcase example is shown in Figure 4. Usually, the test system needs to simulate multiple functional blocks at the same time as parallel test components (PTC). A PTC realizes in fact the functionality of a block from the architecture diagram connected to the selected SUT. Additionally, the test configuration also describes the ports used for communication.
The test data is described in form of signal values. The functional requirement presented in Listing 1 requires that the KeyStatus is “KL15”. The test data for this signal value is provided in Listing 2. The signal KL15 indicates a special setting of the KeyStatus port type used by the test component of type FunctionKey_Type to send that value to SUT over KeyStatus port.

```
signal  KL15  :=  { “KL15” };
type port KeyStatus_PortType {
   out  KL15;
}
type component FunctionKey_Type {
   port KeyStatus_PortType KeyStatus;
}
```

Listing 2. Example of test data

Test behaviors are descriptions of signals applied to input ports of SUT and observations made at the output ports of the SUT. The test sequence for our testcase example is presented in Listing 3. The c-send operation describes a continuous signal applied to a port.

```
Testcase ID: #001
Test components: Function Key, Mirror Side Selector, Mirror Position Setter, Driver Mirror Vertical Control, Driver Mirror Horizontal Control
Test purpose: No vertical movement when Mirror Position Setter is “right”
Precondition: Function Key c-sends KL15 on port Key Status
When: Mirror Side Selector c-sends “DriverSide” on port Mirror Side and Mirror Position Setter c-sends „right“ on port Move Direction
Then: Driver Mirror Vertical Control c-receives “off” on port Motor Control
```

Listing 3. Example of a test sequence
The functional requirement in Listing 1 requires that the mirror on the driver side is moved horizontally to right. The test derivation algorithm correlates the architecture information with the functional requirement and tests that a certain side effect cannot occur. In our example, the test validates that the SUT does not move vertically the mirror in case the mirror position setting button is pulled to “right”.

4.2 Test configuration adaptation to technical view

To reuse test environments that have been derived and designed for the logical view on the technical layer, we might have to introduce adapters between the test components (PTCs) and the SUT (see Figure 5). Those adapt the data formats from the logical specification to the data formats on the technical layer. The information necessary for this adaptation is also available in the architecture model of the system and can be derived from it.

Figure 5. Adaptation between logical view and technical view information formats

5 Conclusions

This paper introduces a new approach for test derivation based on system architecture viewpoints which allows combining the aspects, relations, and models of the various kinds of system components. For testing purpose, the architectural viewpoints open the possibility to derive new types of tests which are not obtained by using other model based testing techniques based on system behaviour only. The architecture
information combined with system behaviour helps to discover system malfunctions caused by interferences or hardware resource sharing.

6 Outlook

The approach can be further refined towards safety and prioritization. The function blocks and the links between them can be assigned with different weights based on their priority or safety critical level. These weights can be then used by the test derivation algorithms to optimize the testing process by limiting the number of tests to the most critical ones only.

References


Model Based Statistical Testing and Time

Frank Böhr ⋆
TU Kaiserslautern
boehr@informatik.uni-kl.de

Abstract. This paper focuses on the introduction of basic time concepts to Model Based Statistical Testing. The concepts presented deal with inputs and outputs of the system under test which have some time dependencies. Three basic time dependencies are introduced: something must happen before a certain time, something must happen after a certain time and something must happen in a given time interval. The approach allows the generation of oracle information considering the introduced time information. This is necessary to test embedded real-time systems. However as far as known to the author nothing has been published concerning the real-time character of embedded systems with respect to MBST.

1 Introduction to MBST

Model Based Statistical Testing (MBST) has been used for testing a wide range of applications. These applications vary from sophisticated software engineering environments [1] to data bases [2] and large industrial software systems [3]. MBST is furthermore used in industries and by government agencies [4]. MBST was also used in projects involved with testing embedded systems such as mass storage devices, weapon systems, medical devices and automotive components [5]. More examples can be found in [6][7][8] and [9]. However, nothing has been mentioned about testing the special characteristics of embedded systems. Embedded systems are very often real-time systems and are becoming more and more complex [10] and increasingly software intensive. Even though additional software adds new features to those systems, it introduces potential failures at the same time. An adequate testing approach must consider the special properties of these embedded systems software (e.g. time constraints). Poore [4] mentions that the software testing problem is complex because of the astronomical number of scenarios and states of use and that the domain of testing is large and complex beyond human intuition and that statistical principles must be used to guide the testing strategy in order to get the best information for the resources invested in testing because the software testing problem is so complex. MBST offers a solution, because a main benefit of MBST is that it allows the use of statistical inference techniques for computing probabilistic aspects of the testing.

⋆ I want to thank The Klaus Tschira Foundation gGmbH and Prof. Dr.-Ing. Dr. h.c. Andreas Reuter for founding and supporting my work.
process, such as reliability [11]. In MBST all possible uses of the software are represented by a statistical model wherein each possible use of the software has an associated probability of occurrence. Test cases are drawn from the sample population of possible uses according to a sample distribution and run against the software under test. The model used to represent the use of the software is a finite state, time homogeneous, discrete parameter, irreducible Markov chain [9]. This model can be represented as a state transition graph as shown in [11] or [12]. The states in this graphical representation determine externally visible states of use. The arcs represent inputs (also called stimuli) to the System Under Test (SUT) and the required SUT response [7]. This means each arc is tagged with a certain stimulus and with a system response which represent possible uses of the system and its reaction. A user might be a human, a hardware device, another software system, or some combination [4].

An example of a graphical representation of a usage model, which represents the use of a telephone, is shown in figure 1. Each usage model has two special states. The first state is called source (On Hook) and the second is called sink (Exit). Each possible walk through the usage model starting at the source state and ending in the sink state is a possible test case [3]. When traversing an arc, the associated stimulus and response are collected. A test case is created by concatenating the stimulus/response pairs of the arcs visited.

Which arc is chosen in a particular state depends only on the sample distribution (shown in brackets in figure 1). So each arc is, additional to the stimulus/response pair, associated with a certain probability. These probabilities are used to generate test cases which are representative with respect to the way in which the software will be used after its deployment. The values of the probabilities of outgoing arcs of each state add up to 1 and depend on the current state only. This is normally a reasonable description of the usage of most software systems [14]. Such a model is called a usage model because it describes the use of the software rather than the software itself. There are several ways to build the structure of a usage model i.e. a graph like the one shown in figure 1 without the probabilities. One way is to apply a method called Sequence Based Specification (SBS). The SBS builds the model structure solely based on the requirements. Approaches introducing hierarchies into usage models are presented in [4][14][15] and an approach dealing with multiple streams of usage is shown in [16].

A usage model can be represented as a Markov Chain (MC). The Markov property of the usage model can be used to compute information which provides a great deal of information about the testing effort [15] like the expected test case length, probability of occurrences for states, long run occupancies of states arcs and inputs [17], source entropy, amount of statistical typical paths and some more [9] prior to any test run and single-use reliability and single-event reliability [17] after the test cases are executed.

According to [2] it is possible to obtain the needed probabilities in three ways. The first is called the un-informed approach. It consists of assigning a uniform probability distribution across the exit arcs for each state. The second one, called the informed approach, can produce many models, and is used when some
Fig. 1. An example of a usage model [13] which represents the use of a telephone.
real usage patterns are available. These patterns could be captured inputs from a prototype or from a prior version of the software. The third approach, called the intended approach, is similar to the informed approach in that it can lead to many models, but the sequences are obtained by hypothesizing runs of the software by a careful and reasonable user. A way which allows some degree of automation in the generation of probabilities is presented in [18].

2 MBST and Time

The mathematical model used in MBST is a finite state, time homogeneous, discrete parameter, irreducible Markov chain [9] with discrete time. In Discrete Time Markov Chains (DTMC) the time between state changes is fixed to a constant discrete parameter $\Delta t$ [19]. So far, however, MBST has neglected the time characteristics of this type of MC. Instead of defining $\Delta t$, MBST counts steps. A test case which traverses $n$ arcs has a duration of $n$ steps. The real time which elapses during a step or the whole test case is not specified within the model. The necessity to consider time when testing an embedded system can easily be seen from the Therac-25 example as described in [20]. Several people were injured or died because increasingly fast user input caused a race condition that triggered a fatal malfunction. The following sections are dealing with the use of the time character of DTMC in MBST.

2.1 Stimulus Occurrence Times

Stimuli are inputs to the SUT. Which stimulus is applicable to the SUT can depend on time constraints. Further consider that the structure of a usage model is derived from specifications [21], and that testing is concerned with showing the correctness of a software with respect to its specifications. This is why the focus of this paper is on time constraints of stimuli which are part of specifications. Three easily imaginable time constrains are e.g. something shall:

1. not happen earlier than $T_1$ time units
2. not happen later than $T_2$ time units
3. not happen earlier than $T_1$ time units and not happen later than $T_2$ time units

after a certain event. These three requirements can be found in an example requirements document of an embedded door control unit which is of realistic size and complexity [22].

Not Earlier Than: Something shall not happen earlier than $T_1$ seconds after a certain event means to wait at least for $T_1$ seconds and possibly longer. This is why not earlier than is split into two parts. The first part deals with waiting exactly $T_1$ seconds. The second part determines the possible additional waiting
Not Earlier Than: A structure which represents the exact waiting of \( n \Delta t \) seconds is shown in figure 2.

The stimulus \( S \) is applied to the SUT exactly \( n \Delta t \) seconds after the event that the stimulus \( E \) was applied because the time spent in a state, each time it is entered, is exactly \( \Delta t \). Most arcs in figure 2 are labeled with \( \omega \) which denotes that no stimulus is applied to the SUT by traversing this arc i.e. waiting. The second part of not earlier than which is the possibly longer part can be represented as a loop, as shown in figure 3.

Not Later Than: Something shall not happen later than \( T_1 \) seconds after a certain event. If it has not happened within \( T_1 \) seconds, it must not happen at all. The distribution of waiting times until \( T_1 \) depends on the SUT. A usage model structure which allows the representation of not later than with an SUT-dependent distribution of waiting times is shown in figure 4.

Basically it is a sequence of usage states that is entered after the event \( E \) is applied to the SUT. This means the first state is always entered thus the minimum waiting time is \( \Delta t \). After entering the first state an arc is chosen based on the probability distribution. It is possible to choose between two arcs as shown in figure 4. One arc is labeled with \( \omega \) (which means waiting), and it ends in the next state where the choice is repeated. The second arc represents a stimulus \( S \) to the SUT. The waiting ends if this arc has been traversed.

Not Earlier and Not Later: The requirement that something shall not happen earlier than \( n \Delta t = T_1 \) seconds and not later than \( m \Delta t = T_2 \) seconds after a certain event can be represented by a combination of the previously shown usage model structures. Waiting at least \( n \Delta t \) seconds is achieved by using the
structure shown in figure 2, and waiting less than \( m \Delta t \) seconds is achieved with the usage model structure shown in figure 4. The realization of the combination is shown in figure 5.

3 How to use the usage model structures

As an example, consider the state Ring Tone of the usage model shown in figure 1. It has one incoming arc labeled dial good and two outgoing arcs labeled connect and hang up. The stimulus connect is associated with a not later than usage model structure. This is the case because it is not possible to get connected after a certain time of \( t_0 \) (\( t_0 = 6 \) seconds in the example below). There will be an error sound if \( t_0 \) seconds are elapsed without getting connected and a ring tone before \( t_0 \). This means the SUT changes its response without the application of an external stimulus. It does this just because the user waits and an internal timer elapses. This is a time dependant response.

The stimulus hang up is associated with a not earlier and not later than usage model structure. This is the case because people usually do not hang up the phone immediately after they dialed a good number and because people usually do not listen to the error sound for a long time after they did not get connected. In this example it is considered, that people do not hang up earlier than 2 seconds after they dialed a good number and not later than 8 seconds. This information about connect, hang up, time and the responses can be represented with the structures shown in figures 4 and 5. It is possible to put the two structures and the information about the two mentioned stimuli together into one combined usage model structure as shown in figure 6. The part which is at the left of the numbered sequence of states corresponds to the not earlier and not later than structure of hang up and the part at the right side corresponds to the not later than usage model structure of connect. The numbered sequence of states itself denotes waiting and the corresponding responses. The structure is entered via the stimulus dial good because the state Ring Tone gets entered via this stimulus. After 1 second it is only possible to get connected or to wait with
Fig. 5. Not Later and Not Earlier: wait between $n\Delta t = T_1$ and $m\Delta t = T_2$ seconds after the event $E$ before the stimulus $S$ is applied to the SUT or do not apply $S$ at all.
a **Ring Sound**. After 2 to 6 seconds it is possible to **hang up**, to get connected or to wait with a **Ring Sound**. The response to waiting changes after $t_0 = 6$ seconds to an **Error Sound** without the application of an stimulus to the SUT. After 6 seconds it is also possible to **hang up** or to get connected. After 7 and 8 seconds it is possible to wait with an **error sound** or to **hang up**. The SUT should start waiting for ever after 9 seconds according to the given time information because the information says their will be no more stimuli. This is why generated test cases end after 9 seconds of waiting after dialing a good number (i.e. go to exit state). It is a good choice to end the generated test cases because it does not change the usage profile, because nothing will happen to the SUT anymore exactly as specified. Each arc in figure 6 labeled with **hang up** ends in state **Exit** because this is the case for the **Ring Tone** state in figure 1. Each arc in figure 6 labeled **connected** ends in state **Connected** accordingly. The **Ring Tone** state of figure 1 can be replaced by the structure shown in figure 6 to introduce the given time information into the usage model of figure 1.

Fig. 6. A Usage Model Structure which can be used to expand the **Ring Tone** state. **Hang up** is abbreviated **hu**, **connect** is abbreviated **c**. The responses of **hu** and **c** are omitted. **r** denotes **ring sound** and **e** denotes **error sound**. $\Delta t$ is chosen to be 1 second.
4 Conclusion

MBST has a high degree of automation. It allows the achievement of fully automated test case generation, test execution and test evaluation based on a usage model [6]. However, it was not possible to consider time constraints in MBST up to the presented proof of concept. This is important in embedded systems testing since embedded systems are usually real-time systems whose proper performance has some time dependencies. It is important to notice that the oracle is generated as well. This means it is possible to decide between pass and fail of a generated test case automatically. This is achieved by using three generic usage model structures.

The presented example shows further on a new possibility to determine a test case end (see section 3). The choice when it is possible to end a test case should be done carefully because it influences the test case length which in turn influences the single use reliability [23]. In the example the approach allows to determine the test case end based on the given time information and hence allows to calculate an appropriate single use reliability.

In addition, the concepts presented allow the incremental development of usage models as shown in the phone example of figure 1. The usage model was initially developed without any consideration of time. With the presented approach it is possible to add time information to the usage model if it is required as shown in section 3. This can be used to test an early version of the software without time considerations. Timing of the SUT might get more and more important in later software versions. At this point of the software development it is possible to add the time information to the usage model to ensure a proper testing of timing requirements.

The next step is evaluating the method using an example of realistic size like the one presented in [6]. Currently a graphical representation is under development, which allows the automatic generation of structures like the one presented in figure 6 together with the associated probabilities. The aim is to be able to generate time dependant usage model structures for different $\Delta$'s and thus to take the automation a step further.

References

Test Case Generation for Embedded Automotive Systems: A Semantics Preserving Model Transformation

Valentin Chimisliu1, Christian Schwarzl2 and Bernhard Peischl1

1 Inst. for Software Technology, Techn. Universität Graz, Graz, Austria
chimisliu@ist.tugraz.at, peischl@ist.TUGraz.at
2 AreaE, Virtual Vehicle Competence Center
Das virtuelle Fahrzeug - Forschungsgesellschaft mbH, Graz, Austria
christian.schwarzl@v2c2.at

Abstract. A well-founded testing theory encourages the practical application of test case generation techniques. This aims at overcoming the ever increasing complexity of software-enabled systems in the automotive industry. In this article we report on transforming UML Statecharts to the formal language LOTOS. Both, the successful usage of UML Statecharts in our industrial setting and the availability of mature research prototypes for test case generation supporting LOTOS suggest this transformation. Our novel transformation manages to preserve the semantics of the UML Statechart, allows for treatment of UML-like events and preserves the atomicity of transitions in the UML Statechart. Moreover, we present first results on the test case generation for our industry example.

1 Introduction

This article reports on an applied research project by introducing test case generation techniques in the automotive industry. One of the project goals is to establish a scalable test generation technique that allows various test strategies (e.g. coverage-based, scenario-based, random or fault-based). Another goal is to integrate the test generation technique with the existing test automation environment, the employed modeling paradigms and the test engineer’s domain experience.

We have decided to harness the well-founded testing theory behind IOLTS (input-output labeled transition systems) and rely on the mature research prototype TGV [1] from the CADP (Construction and Analysis of Distributed Processes) toolbox. TGV allows for on-the-fly exploration of the state space for the provided specification thus, offering a way of avoiding the construction of the whole state space. The primary input language of TGV is the ISO standardized language LOTOS [2] (Language of Temporal Ordering Specification). Although LOTOS has directly been applied in the industry [3], the pragmatics of the language, the lacking support from industry production tools, and the need for getting used to its modeling paradigm underpin our industry partner’s needs to rely on the UML statechart formalism.

We have decided to use the CADP toolbox because of its open architecture, sound verification theory, and limited number of domain specific assumptions as opposed to other tools available on the market.
In this article we report on the challenges in providing a semantics-preserving model transformation obtained on an industry example.

Section 2 presents our industry example for the HiL environment. Section 3 gives a brief introduction to the LOTOS language and its underlying modeling paradigm while in section 4 we introduce the transformation rules used to derive the LOTOS specification from the UML statechart. Section 5 contains first experimental results obtained by applying a coverage-based test generation strategy on the stated industry example. In section 6 we discuss related research and section 7 concludes the paper.

2 Model Description

Statechart diagrams are used to visualize state machines and belong to the category of UML behavioral diagrams. As the statechart formalism is well known, we will not go into details on this topic. Detailed descriptions of this type of diagrams can be found in the many books written about UML or in the UML standard [4].

The behavior of the considered system is described by means of UML models [4]. The used UML model describes the diagnosis functionality in modern vehicles. Its purpose is to store the type, occurrence and origin of errors. This information is stored in a dedicated buffer and is subsequently used by mechanics to localize and repair/replace damaged ECUs, sensors or actuators.

The Error Handling Model defines how a detected error is treated. It specifies which conditions have to be fulfilled to create an entry in the error memory.

In Fig. 1 the model of the diagnosis functionality is shown. It consists of five states and accepts only four messages namely evErrorActive, evErrorNotActive, evRequestErrorMemoryClear and evSetGwUnlearnCounterCopy.

The state S_NotActive_NotStored corresponds to normal functioning when no error has been detected. After an error is detected, the system moves to the state S_Active_NotStored which means that an error has been detected but is not yet stored. The error is stored after receiving five evErrorActive events and the system moves to the S_Active_Stored state. This means that the real ECU has now stored an error in its buffer which can be read out with dedicated diagnosis hardware. The diagnosis module shall leave this state and move to the S_NotActive_Stored state only after receiving an evErrorNotActive event.

3 LOTOS Introduction

LOTOS [2] is a formal description technique developed within ISO for the formal specification of open distributed systems. LOTOS is composed of a process algebraic part based on Milner’s Calculus of Communicating Systems (CCS) and on Hoare’s Communicating Sequential Processes (CSP) [5], and a data algebraic part based on the abstract data type language (ACT ONE).

A LOTOS specification describes a system as a hierarchy of process definitions. Thus, a system can be modeled as a process that may contain several subprocesses. The LOTOS model of a system is viewed as a black box with a number of gates that can be seen from its environment.
The behavior of a process is specified through behavior expressions composed of gate offerings and LOTOS operators. The operators are used to combine behavior expressions in order to form more complex behavior expressions. In the remainder of this section, behavior expressions are denoted using upper case letters and the gate names as lower case letters. As detailed information on the language can be found in [2], below we present only some of the most important LOTOS operators:

- The sequentiality operator “;”, called action prefix composes an action $g$ with a behavior expression $B$. The composition is another behavior expression describing a system that will initially accept action $g$ behaving afterwards as $B$.
- Choice “[]” composes two alternative behaviors describing a system that offers to the environment two (or more) alternatives. An example would be: (Car\_started; DRIVE) [ ] (Car\_broken; WALK)
- The full Synchronization: “|||” operator denotes the fact that the events which occur in either of the behavior expressions have to synchronize. The expression $(a,b,X)|||(a,b,Y)$ may engage in the sequence of events $a,b,...$.
- The interleaving operator “|||” allows behaviors to unfold completely independently in parallel; the events from each behavior expression are interleaved. The behavior expression $(a;bc;P)|||(xy;T)$ includes the behaviors: $ax;xy;bc...$ and $a;bx;xc;y...$, etc.
– **Partial Synchronization:** “|[< gates >]|” means that concurrent behaviors synchronize on the gates listed in the operator. Thus, events occurring at gates in the list will synchronize while the ones occurring at gates not in the list will interleave. The behavioral expression \((a;bc:P) \parallel (b;y;T)\) offers the behaviors \(a;b;cy;\ldots\) and \(a;bc;y\ldots\).

– **Inaction:** “stop” represents a system that cannot show any action. It can be used to express the fact that a process has finished its execution or to express explicit deadlock situations.

– **Successful Termination:** “exit” represents the successful termination of a behavior expression.

### 4 Model Transformation

This section describes the transformation rules used to derive a LOTOS specification from an UML statechart description of a system. It is assumed that the UML model is well formed according to the criteria defined in [4].

The first step of the transformation is that the variables used inside the statechart are mapped to LOTOS process parameters while the events are mapped to gates. Every state in the statechart is mapped to a LOTOS process.

#### 4.1 Transforming Composite States

For each composite state [4] a new LOTOS process is created. For every composite state that does not have a nested history node, the resulting process definition is presented below. The description uses the Extended BackusNaur Form.

```plaintext
process "STATE_ID" "|" (< gate_list >) "|" (< param_list >) "|" noexit "=" (< substate_id >) "|" (< gate_list >) "|" (< param_name >) "|"
endproc
```

Here `<gate_list>` is the list of gates corresponding to the events in the statechart, `<param_list>` are the parameters to which the variables in the statechart were mapped to and `<substate_id>` is the name of the process generated for the substate targeted by the default transition [4] in the composite state.

For every History pseudostate in the statechart, a new process parameter is inserted in the LOTOS specification. The inserted parameter is used to keep track of the last active state in the statechart. Thus the resulting process definition of a composite state containing a History pseudostate is presented below.

```plaintext
process "STATE_ID" "|" (< gate_list >) "|" (< param_list >) "|" noexit "=" (< behavior_expression >)
endproc
```

Where `<gate_list>`, `<param_list>` and `<behavior_expression>` are defined in terms of:
In the definition of $s_id$, $n$ denotes the number of states nested inside the composite state. For simplicity reasons it is assumed that the name of the gate $GATE_i$ also contains the possible parameters of the events in the statechart mapped as gate events.

In the presented Diagnosis model, the state $S_{Diagnose Model Message}$ is a composite state and contains a History node.

Example 1

```latex
\begin{verbatim}
process $S_{Diagnose Model Message}[gates](\text{parameters list})$: noexit:=

[HST1 = 1] -> DM_S_NotActive_NotStored [gates](\text{parameters})

[HST1 = 2] -> DM_S_Active_NotStored [gates](\text{parameters})

\end{verbatim}
```

4.2 Transforming Simple States

According to [4] completion transitions shall be fired as soon as their guard evaluates to true. Thus the statechart will accept an event only after reaching a stable state. This behavior is referred to as the run to completion step.

In order to model this kind of behavior, a conjunction of the negated guards of the completion transitions is added as a guard that restricts the behavior in the resulting LOTOS process. The obtained behavior is that the gates corresponding to non completion transition are offered to the environment only if the added guard evaluates to true (no completion transition can be fired).

The conflicts between transitions that are fired by the same event and are at different nested levels in the statechart are resolved during the transformation in a manner similar to the completion transition issue presented above.

The behavior of such processes is obtained by transforming the transitions originating from the state into LOTOS behavioral expressions resulting in a
process like the one below. The transformation of the transitions into LOTOS expressions are described in section 4.3.

```process "STATE_ID""["<gate_list>"""]""("<param_list>""")""""noexit""=""{
""[not""<compl_guard> {""and"" ""not""<compl_guard> } ]"" -> "
<gate_name> {""[""<ev_guard> {""and"" ""not""<ev_guard> } ]""}
<state_id> {""[""<gate_list>"" ]"" {""<param_list>"" }}
{""[""}<gate_name> {""[""<ev_guard> {""and"" ""not""<ev_guard> } ]""}
<state_id> {""[""<gate_list>"" ]"" {""<param_list>"" }}
""]""
{""[""}<compl_guard> {""-> ""<gate_name>""}"
<state_id> {""[""<gate_list>"" ]"" {""<param_list>"" }}
}

endproc
```

The “COMPL_GUARD,” and “EV_GUARD,” in the `<compl_guard>` and `<ev_guard>` defined below represent the guards of the completion transitions and the ones on the other transitions, respectively.

```<compl_guard> := “COMPL_GUARD1” | ... “COMPL_GUARDn”
<ev_guard> := “EV_GUARD1” | ... “EV_GUARD2” | ... “EV_GUARDn”```

4.3 Transforming Transitions

The transformation assumes the derivation of a flattened representation of the statechart. Every direct path between connected states is first constructed. This is done by converting all compound transitions originating from the current state into transitions that target other states (as opposed to pseudostates). For every transition originating from a junction pseudostate, a new transition is created containing the triggering event of the transition targeting the junction and the conjunction of the guards of both transition segments (targeting and originating from the junction pseudostate).

In the case of choice nodes, the same process applies. The only difference is that the added guard needs to take into account the changes (to the variables) made in the current run to completion step. In [4] this behavior is called a dynamic choice.

After the transitions have been computed and they target states as opposed to pseudostates every transition is transformed into a LOTOS behavioral expression. Due to the fact that all components of a transition (event, guard, effect) are optional, the resulting behavioral expression can take several forms.

The behavioral expression corresponding to a transition has the following form:

```<trans_behexp> := {<gate_name>} {""[""<guard>"" ]"" ; ""}
{<gate_name> ; ""}
<state_id> {""[""<gate_list>"" ]"" {""<param_list>"" }}
```

48
In the expression above the first \(<gate name>\) represents the triggering event of a transition and \(<guard>\) is the mapping of the guard of the transformed transition. The effect part of a transition is represented as follows: if the transition generates an event, this is mapped to the second \(<gate name>\) in the above expression, while the operations on the variables in the statechart are mapped to the parameters of the process \(<state id>\).

The state \(\text{DM}_S\text{NotActive}_\text{NotStored}\) from the Diagnose model is transformed into the process:

**Example 2**

```plaintext
process \(\text{DM}_S\text{NotActive}_\text{NotStored}\)[gates](parameters): noexit:=
  evErrorActive !\text{DM}_\text{trans}_3 ?id:\text{Nat} [id = mdl];
  DM_S_Active_NotStored [gates](parameters)
[
] evReqErrorMemoryClear !\text{DM}_\text{trans}_2 ?id:\text{Nat}
 [id = mdl];
  DM_S_NotActive_NotStored [gates](parameters)
......
endproc
```

Where the \(!\text{DM}_\text{trans}_i\) are LOTOS gate events used to provide a way to identify the transitions of the statechart in the derived LOTOS specification.

In the proposed transformation the events in the state chart are mapped to gates in the LOTOS specification. The atomic nature of firing transitions is preserved through the fact that once the first action of a transition is offered to the environment, the only allowed sequence of actions is the one representing the rest of the actions on that transition.

**5 Experimental Results**

One of the goals behind this transformation is to allow for the usage of already existing test generation tools to automatically extract test cases from UML models. There are already several test case generation tools in the academic community that accept LOTOS as an input language for specifying the behavior of systems. One of the most mature tools of this kind is TGV (Test Generation from transitions systems using Verification techniques), which is integrated into the CADP toolbox (see http://www.inrialpes.fr/vasy/cadp for details).

TGV uses the concept of test purposes to focus the test case generation and to avoid the state space explosion problem. Test purposes have to be manually formulated. The quality of the resulting test cases strongly depends on the skills of the tester - in terms of specifying appropriate test purposes.

Significant effort has been put into the development of techniques to automate the test purpose generation process. Such a technique [6] was used on the Diagnosis model to automatically derive test purposes and subsequently test cases. The results are presented in Table 1.

The first column of Table 1 contains the coverage criterion for which the test cases were generated. The second column presents the number of generated test
purposes. In the next column, the total time in seconds needed to process these test purposes is shown. The last column lists the number of generated test cases.

Table 3 lists the statistics regarding the test cases obtained for process coverage. The first column contains the number of the test case and the second one the number of covered processes (corresponding to the states of the statechart). The next two columns list the number of states and transitions of the IOLTS describing the test case. The last column contains the time needed to compute the test cases.

When using this method, some of the obtained test cases were very lengthy. For example test case number 5 covering five states of the Diagnosis model ended up having 1120 states and 1119 transitions. One reason for this is the high branching factor of the IOLTS generated from the LOTOS specification and the fact that TGIV uses a depth first search algorithm to search the IOLTS.

A proposed solution to reduce the length of the generated test cases is to use a breadth first search algorithm to search the IOLTS. The applied method [6] is based on the insertion of probes into the LOTOS specification. We have used the model checker Evaluator from the CADP toolbox to search the IOLTS using a breadth first search algorithm this way guaranteeing the shortest path to the inserted probe. The results obtained with this approach are summarized in table 2 while table 4 lists the statistics of the test cases generated for process (states in the statechart) coverage. Thus the IOLTS describing test case no. 5 covering five states ended up having 7 states and 6 transitions.

There are still issues to be resolved regarding the fact that part of the generated test cases contain redundancies (considering the coverage criteria). However, the obtained results indicate that the proposed model transformation could work in an industrial setting.
6 Related Work

There have been several approaches to provide a formal semantic to both structural as well as behavioral aspects of the UML [7]. Approaches of how UML statecharts can be verified using the model checker SPIN [8] can be found in [9].

Closer to the technique presented in this paper is the one described in [10], which defines mapping rules for some of the structural and behavioral aspects of UML. Concerning the behavioral aspects, in [10] the focus is on the transformation of activity diagrams to LOTOS. In [11] and [12] transformation rules from a UML statechart to LOTOS are presented. However, the transformation in [11] has several limitations, the most important ones of which are: considering only basic LOTOS (i.e. without allowing the use of variables in the statechart), considering only normal states and does not allow the crossing of the borders of composite states.

Significant effort has already been invested in the automatic generation of test suites from models of the system under test. One such approach was developed in the AGEDIS project [13] that used the AGEDIS modeling language as input. The test generation engine used in this project combines the principles of TGV [1] and GOTCHA [14].

The approach presented in this paper aims at easy integration in the already existing development process and the used tool chain in industry projects.

7 Conclusions

In this article we report on an applied research project dealing with the introduction of test case generation techniques in the automotive industry. One project goal is to establish a scalable technique that allows for the application of coverage-based, scenario-based, fault-based and random testing strategies. In our specific project setting it is of utmost importance to provide a solution that can easily be integrated into the test automation set-up in place (a HiL test bed), the employed modeling paradigms (i.e. UML statecharts), and the test engineer’s domain expertise. In this article we propose a semantics-preserving model transformation from UML Statecharts to the specification language LOTOS - the primary input language of mature research prototypes for test case generation.

The obtained results indicate that the derived models allow for deducing meaningful test cases with a reasonable computational effort in a fully automated way. As states in the UML model map to LOTOS processes our transformation allows for coverage-based test case generation. Furthermore, our results point out the need for high quality test purposes as they allow for considerable reductions of the state space of the finally obtained IOLTS.

Future work includes improvements to the transformation algorithm regarding the number of treated components of the statechart formalism. Another goal is to allow the transformation and respectively test generation from specifications describing the behavior of asynchronous communicating concurrent models.

Further research is needed to improve the quality of the generated test purposes in order to eliminate redundancies in the obtained test cases. Also more effort is required to present the specification of test purposes in a way suitable
for use in industrial projects. Describing a test purpose with the help of sequence charts is one possibility of addressing this problem.

Acknowledgement

The authors wish to thank the “COMET K2 Forschungsförderungs-Programm” of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economics and Labour (BMWA), Österreichische Forschungs- und Entwicklungs-gesellschaft mbH (FFG), Das Land Steiermark and Steirische Wirtschaftsförderung (SFG) for their financial support.

References

10. da Silva, P.P.: A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, Manchester, UK (June 2001)
SESSION 2
UML-based MBT Approaches
Transformational Support for Model-Based Testing – from UML to QML

Fredrik Abbors1, Tuomas Paijärvi1, Risto Teittinen2, Dragoș Trușcan1, and Johan Lilius1

1 Dept. of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5, 20520, Turku, Finland
   Email: fabbors|tpaijarvi|dtruscan|jolilius@abo.fi
2 Nokia Siemens Networks, Linnoitustie 6, 02600, Espoo, Finland
   Email: risto.teittinen@nsn.com

Abstract. Model-Based Testing (MBT) has lately gained increased popularity due to the benefits that it provides in terms of automation of the test generation process. There are several tools capable of applying MBT using behavioral models of the system under test (SUT). However, as complex systems are specified using different perspectives, like architecture, data, behavior, which benefit from proper tool support especially in the Unified Modeling Language (UML) community, there is a gap between the graphical capabilities and expressiveness of the UML-based and MBT tools. In this paper, we present an approach in which the information describing different perspectives of the system under test is collected from the UML models of the SUT and transformed into input for a MBT tool, to be used for automatic test generation, execution, and evaluation.

1 Introduction

Testing has become an important activity of the software development, which according to some studies can take more than 40 percent of the development resources [1]. Recently, Model-Based Testing (MBT) has gained momentum by advocating the use of models for automatic test generation, execution, and evaluation. The basic idea in MBT is to check the conformance of a system implementation, aka system under test (SUT), against a model specifying its behavior. Currently there are established approaches for performing MBT from behavioral models, and the corresponding tools are available (an exhaustive list can be found in [2]). However, complex systems are often described using several perspectives like behavioral, architectural, data, which should also be considered during the testing process. Thus, one should include such information in the specification of the SUT and use it for test generation. Most of the current MBT tools provide support for modeling the behavior of the SUT, but few are able to consider different types of information or the information is expressed using textual format.

In recent years, the Unified Modeling Language (UML) [3] has become one of the most popular languages for system specification. UML provides a collection of diagrams that can be used to model different perspectives of the system and there is a plethora of tools that can be used for UML-based specification. Therefore, we would like to take advantage of the existing UML tool support for specifying the SUT, and later use the
resulting models for test generation. In order to have a smooth transition between the UML and the MBT tools, transformations should be used for collecting necessary information and creating the input for the testing tools. Such transformations will also enable a smooth integration of the UML tools in the MBT tool chain.

In this paper we employ a transformational approach to reduce the gap between UML and MBT tools. As an instance of our approach, we exemplify with a concrete transformation that translates UML models into input for a tool used for automated test design, namely Conformiq’s Qronic [4]. The transformation is defined in a general manner with respect to UML, such that it can be applied to different UML modeling tools. The approach is exemplified with a telecommunications case study.

1.1 System Modeling with UML

In our system specification, we apply a systematic approach for creating a set of models from the requirements in order to capture different views of the SUT. UML is used as specification language. The main purpose of the modeling process is to describe several perspectives of the SUT, like requirements, architecture, data, behavior, such that the information contained in these models can be used not only for development, but also for automated test generation using specialized test generation tools. As the models are derived from requirements, it is important to track how different requirements reflect in the models, on different perspectives and on different abstraction levels. It is also important to propagate requirements through the test generation and test execution processes, such that one can verify what parts of the models and consequently, which requirements have been tested and validated.

Our approach is divided into five (horizontal) phases as depicted in Fig. 1. The first phase deals with requirements analysis, and has as main purpose the identification of the system requirements from the related standards and protocol specifications. The second phase structures the features and functional requirements of the system, using the UML and Systems Modeling Language (SysML) [6] diagrams, respectively. In the third phase, the main usage scenarios of the system are specified in a use case model both textually and via message sequence charts (MSCs). The fourth phase looks into deriving the domain, behavioral and data models of the system starting from the previously specified models. The process is iterative so phases three and four can be visited several times and the models are constructed

---

3 detailed examples can be found in [5]

---

Fig. 1. System modeling process
incrementally. The last phase deals with creating specific system configurations by instantiating the concepts abstracted in the precedent models. Several models result after applying this process and they are described in more detail in the following sections.

A set of rules and guidelines are defined for ensuring the quality of the resulting models, by checking that the models are syntactically correct [7]. These rules also ensure that the models are consistent to each other and moreover, that they contain the information needed in the later phases of the testing process. Tool support is provided for automatically verifying these rules using the Object Constraint Language (OCL) [8].

1.2 The Qtronic tool

Conformiq Qtronic [4] is a tool for model-driven test case design, which generates tests from the specification of the SUT using various coverage criteria. The Qtronic Modeling Language (QML) is used for describing different aspects of the SUT like input/output ports and messages, message structure, behavior, etc. The behavior of the SUT can be described either textually in Java or graphically using UML state machines. State machines are drawn with the Qtronic Modeler, whereas the rest of the system specification is done in QML, following object-oriented principles. As such, the SUT is specified as a class that can have attributes and methods. The methods are later used as the action language in the state model. In Qtronic, the messages sent or received by the SUT are defined as records, that is user-defined types that can contain variables, methods, operators, and nested types. Records can inherit other records. Inbound and Outbound ports describe which records may be sent or received. Several input/output ports can be specified. The ports and the messages that are exchanged constitute the interface of the SUT to its environment. This interface is declared under the system-block. In QML, a state machine is a separate thread. Other threads can be declared by extending the Thread-class or by declaring a class that implements the Runnable interface. One important aspect is that multiple instances of the same state machine can be executed in parallel in order to allow testing of concurrent behavior. The communication between threads can be implemented either by message passing or by shared variables.

2 Deriving the QML Model from UML

The main purpose of the transformation is to automatically extract the necessary information from the UML models and use it for creating the model used by Qtronic for test generation. The transformation can be seen as composed of several parts (see Fig. 2), each described in the following subsections. Throughout this section, we will use excerpts from a telecommunications case study modeling a Mobile Switching Server (MSS), which will be used as the SUT. The MSS is a central element of 2nd and 3rd generation mobile telecommunication networks. The MSS connects calls between mobile
phones and fixed network. It takes into account movement of mobile phones at the time of call set up and during the calls.

2.1 Generating the Interfaces and Ports of the System

In our UML modeling approach, a domain model (Fig. 3) is used to depict the components of the domain, and how they are connected via interfaces. Domain components can communicate with each other via messages belonging to various protocols. For cutting down the complexity, the messages sent and received on each protocol level, for instance Mobility Management (MM), are modeled separately by interface classes.

In QML, interfaces are described with the system-block which describes ports that can be used to communicate with the environment, and what messages on each port can be sent and received. The Inbound ports declare messages to be received by the SUT from the environment, whereas the Outbound ports declare messages to be sent from the SUT to the outside world. As such, the ports of the SUT are obtained directly from interface classes in the domain model (see Fig. 3). Inbound messages are taken from the interface realization offered by the SUT, and Outbound messages are taken from the interface realization used by the SUT. The name of the ports will be composed of two components, the direction and the interface name, respectively. UML operations are listed as messages transferred through the ports, and they are declared elsewhere as records. Section 2.2 describes in more detail how QML records are created and how record variables are defined. The partial result of applying the transformation on the MM interfaces in Fig. 3 is shown in Listing 1.1.

Listing 1.1. Example of QML system-block for Fig. 3

```qml
Listing 1.1. Example of QML system-block for Fig. 3
```
2.2 From UML Data Models to QML Message Types

In order to generate proper test cases from system models, a description of the data used by the system is needed. We model these data explicitly via class diagrams. We refer to this model as the data model. The data model of the SUT depicts each message type as a class, whereas the parameters of the message are represented as class attributes. We structure the message definition based on their corresponding protocols and use inheritance to model common parameters for a given message. Figure 4 presents an example of a UML data model from our case study.

![Fig. 4. Message declaration in UML.](image)

In QML, messages are described as records that are used for communicating with the environment. QML records are user-defined types similar to classes. The fields of a record can be of type: byte, int, boolean, long, float, double, char, array, String, or of another record type. In our transformation, records are obtained from classes in the UML data model. Attributes of the UML classes are transformed into the fields of the record. Inheritance in UML is reflected in QML using the extends relationship. For instance, the location_update_request record in Listing 1.2 is obtained from the LOCATION_UPDATE_REQUEST class in Fig. 4 following the described approach. We point out that the model does not indicate value ranges of the fields. Instead, the value ranges are checked later on by the protocol codecs provided by the test system.

### Listing 1.2. QML record declaration

```qml
record MM_messages {
  public String protocol_discriminator ;
  public String skip_indicator ;
  public String message_type ;
}

record location_updating_request extends MM_messages {
  public int location_updating_type ;
  public String ciphering_key_sequence ;
  public String location_area_identification ;
  public String mobile_station_classmark1 ;
  public String mobile_identity ;
}
```

2.3 Mapping the UML State Machine to the QML State Machine

As mentioned previously, the behavior of the SUT can be specified in Qtronic either textually in QML or graphically using a restricted version of the UML state machines. For simplicity, we have chosen to follow the latter option as the target of our transformation. Thus, the transformation is basically a matter of transforming the UML state machine into the corresponding state machine used by the Qtronic tool, which in practice is equivalent with a transformation at the XMI-level. Figure 5(a) shows a UML state machine\(^4\), whereas Fig. 5(b) shows the same state machine transformed to QML. As one can notice,

\(^4\) The state machine has been created in the MagicDraw UML modeling tool

---

59
there is a strong similarity between the two models, however with small differences. For instance, both state and sub-state machines are supported and propagated at Qtronic level. In UML, triggers and actions are declared as methods (selected only from the operations of the interface classes in the domain model).

In QML, triggers are implemented by messages (record instantiations) received on a certain port, whereas actions can be seen as methods of the SUT class definition. This approach allows one to perform further processing of the system data before sending a given message to the output port\(^5\). The method generated for the \texttt{MM\_LOCATION\_UPDATING\_ACCEPT()} in Fig. 5(b) is shown in Listing 1.3.

\subsection*{2.4 Propagating System Requirements from UML to QML}

In our modeling approach, requirement models are used for structuring and interrelating the requirements of the SUT using SysML Requirement Diagrams. The requirements are defined on several levels of abstraction following a functional decomposition and they can be related to other requirements or linked (traced) to different diagrams and model elements in UML, for example to state machines and class diagrams. Figure 6 shows an example of a requirement diagram. The main purpose of tracing requirements is in analyzing which parts of the specification “implement” different requirements and it will allow later on to propagate these requirements from models to tests.

Qtronic provides support for requirement coverage during test generation. Requirements are associated to state models, more precisely to the actions on transitions via the \texttt{requirement} statement. This is achieved by either attaching a requirement as an action on a transition, or in a method in the QML textual notation file. Basically, the re-

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5.png}
\caption{Example of UML state machine (a) and its equivalent in Qtronic Modeler (b)}
\end{figure}

\begin{verbatim}
Listing 1.3. Example of a generated QML method
\begin{verbatim}
void MM\_LOCATION\_UPDATING\_ACCEPT()
{
    MM\_LOCATION\_UPDATING\_ACCEPT location\_updating\_accept;
    MM\_out.send(location\_updating\_accept);
    return;
}
\end{verbatim}
\end{verbatim}

\footnote{If needed, additional QML instructions can be manually inserted in the generated methods, before the \texttt{.send()} statement}
requirements in Qtronic are tags that are used to trace if a specific action in the state model has been covered by the generated test cases.

In the current approach, we are interested in collecting from the UML state models only those requirements that are attached to state transitions. However, nothing prevents us from collecting the requirements from other UML diagrams, as well, if needed. The requirements are captured from UML transitions and placed on the corresponding transition in QML, using the requirement statement (see Fig. 5(b)). Hierarchy can also be propagated from UML to Qtronic by analyzing how the requirements are structured in the requirement models. For instance, requirement “6/1/1” in Fig. 5(b) is obtained from requirement “6.1.1” in Fig. 6, which at UML level was attached to the transition Authentication → Ciphering in Fig. 5(b). Upon importing a QML model in Qtronic, requirements are displayed hierarchically in Qtronic’s UI (Fig. 7) from where requirements-based test derivation can be pursued.

2.5 Generating the QML Test Configuration

In UML, we use object diagrams (see Fig. 8) for specifying network configurations instantiated from the domain model (see Fig. 3). Multiple instances of the same class can be instantiated and properties specific to each instance are initialized. Such diagrams can be used for defining the testing setup in the QML model. In this case the :MSS depicts the SUT, while the other instances represent the test environment.
As illustrated in Figs. 3 and 8, a \(MS\) (mobile subscriber) is a user of the SUT and communicates with the SUT via the MM interface. The test configuration in our example consists of three mobile subscribers. In QML, a subscriber is modeled as a record (Fig. 9(a)) with the same attributes as in the domain model (Fig. 3). The test configuration is translated into QML in two steps: first the properties of the test components are extracted from the UML domain model and are declared in the constructor method of the SUT specification class (MSS in our case) as shown in Fig. 9(b). In the second step, each test component is instantiated and its properties are initialized with the values taken from the configuration diagram (Fig. 9(c)). The test components are stored into an array, which is later used for starting as threads separate versions of the SUT.

2.6 Assigning the State Model to the SUT Specification

At this point, the only thing left to be done is to connect the SUT class specification to the graphical state model. This is done by calling the constructor method for the state machine. Once the state machine has been constructed, the concurrency of the SUT can be tested by starting (via the \textit{Thread.start}() method) separate execution threads for each test component (i.e., subscriber) (Listing 1.4). The approach allows for different mobile subscribers to concurrently communicate to the SUT using different configuration parameters. This is needed as in telecom systems such as MSS one needs to test the presence of multiple mobile subscribers interacting with the MSS (in practice, one MSS can serve up to few millions of users). In addition, we need to test calls between pairs of subscribers, where one call requires two subscribers, i.e. the one who calls and the one who receives the call (known as A and B subscribers).

3 Tool Support

The mappings from UML to QML discussed in the previous section have been defined in such a way that any set of models based on UML and containing the required information can be used as a source of the transformation. However, as various UML tools implement the UML metamodel differently, the implementation of the transformation, from UML to QML, had to be customized for the specific tool used.

In our case, the MagicDraw [9] modeling tool has been used for creating (via the process described in Fig. 1) and for assuring the quality of the UML models via validation
rules written in OCL. Once completed, the models are exported to XMI [10] format. A Python script is used for automating the transformation. The script consists of seven separate modules (Python-source files). A parser module parses the XMI file and structures the information in a tree-like representation (using the lxml [11] library). This representation is used by the other modules for creating QML-related information. A state machine creator module uses the collected information in the parser to generate the QML state machine. A record creator module uses the collected information in the parser to generate the QML records. A method creator module uses the information collected by the parser to generate the textual annotations for ports, methods, and for the main function. A definition module declares patterns of the QML elements that are used for creating the QML models. A manifest creator module generates the QML manifest file, which lists the generated files for a given project, whereas a main module is used to control and to invoke the functionality of the other modules during the transformation.

The script is structured in such a way that it easily can be modified and extended to suite input from other modeling tools or generate models for other MBT tools. For example, if another modeling tool is used, only the parser module has to be modified.

4 Related Work

Several research works discuss transformational approaches for model-based testing. In the following we will only discuss several of them which we consider related to our approach. A transformation from test models, expressed using the UML 2 Testing Profile (U2TP), to TTCN-3 is proposed in [12]. This transformation is closer to code generation and has as the main purpose the creation of test specifications from test models, while our transformation translates UML models into input for a test derivation tool. A transformational framework in the context of MDA is suggested in [13] with the main purpose of deriving tests from system models expressed in a restricted version of UML, called essential UML. A set of transformational rules are specified between the source and the
target language, the essential Test Modeling Language (eTML), a variant of the U2TP. A similar approach is described in [14], where test models are integrated along with a MDA development process. Again, the main focus of the transformation is on deriving the test specifications expressed in TTCN-3 at different levels of the MDA process. The source models for the proposed transformations are expressed using a UML profile for Enterprise Distributed Object Computing (EDOC) or in Sun’s J2EE. This work is closer to ours since it generates the final test specifications by combining model transformation and code generation.

5 Conclusions

We have presented an approach in which the UML models used for system specification are automatically transformed into input for a test generation tool. Our work had two main goals: on the one hand, we wanted to show that one can benefit from the graphical capabilities and expressivity of the UML specifications during the test generation process. Such an approach circumvents the currently limited modeling capabilities of certain test generation tools. We also wanted to take advantage of the validation capabilities of existing UML tools, which currently exceed those of the MBT tools. On the other hand, we intended to provide a solution for improving the transition between different steps of the MBT process, by suggesting a set of mappings between UML and QML that can be used to automate the transition and thus improve the existing tool chain.

References

11. lxml homepage: http://codespeak.net/lxml/.
Combining Sequences and State Machines to Build Complex Test Cases

Dehla Sokenou¹ and Stephan Weißleder²

¹ GEBIT Solutions GmbH, Koenigsallee 75 b, 14193 Berlin
dehla.sokenou@gebit.de
² Humboldt-Universität zu Berlin, Rudower Chaussee 25, 12489 Berlin
weissled@informatik.hu-berlin.de

Abstract. Model-based testing is an important testing technology. The UML is a very popular modeling language. In this paper, we present a technique to utilize UML state machines in order to concatenate UML sequence diagrams and sketch corresponding coverage criteria. We show the relevance of our work by presenting an industrial case study in which sequence diagrams were combined with a state machine.

Key words: Unified Modeling Language, State Machines, Sequence Diagrams, Test Model Combination

1 Motivation

Functional testing is an important testing technique that is based on comparing the system under test (SUT) to specifications. Model-based testing is often used for functional testing. The test models are used as specifications. They provide information about input stimuli as well as the necessary oracle to deduce if a test case detects a fault.

Functional testing compares SUT and test models. Thus, SUT and test models must not be automatically derived from another and the creation of test models is often a manual task. In model-driven engineering, a requirement use case is often supplemented with a sequence diagram. Each sequence diagram represents a few possible behaviors of the SUT. Thus, there are only a few test cases for each use case. This supports traceability from requirements to test cases. For such reasons, sequence diagrams are very popular to model test cases. One issue about sequence diagrams, however, is that they just consist of a sequence of interactions without any notion of state. Thus, it is impossible to concatenate sequence diagrams, i.e. execute the described test cases consecutively.

State machines are a more complex means to model behavior than sequence diagrams. In contrast to a sequence diagram, a state machine is used to describe a large set of behavior traces: From most non-trivial state machines, a possibly infinite set of test cases can be derived. In such cases, coverage criteria are used as a stop criterion for test generation. Since the described behavior can be quite complex and the generated test cases are also determined by the used coverage
criteria, the application of state machines makes traceability hard. However, state machines are based on state information, which can be utilized, e.g., to derive test oracle information based on state invariants.

In this paper, we propose the combination of state machines and sequence diagrams. The contribution is a technique to concatenate sequence diagrams by retracing their behavior in state machines and concatenating the corresponding transition sequences. For that, all possibly traversable transition sequences of the state machine matching to the information contained in the sequence diagrams have to be identified. The advantages are, e.g., the creation of longer sequences based on existing ones, the detection of faults that are undetected by the tests derived from simple sequence diagrams, or the satisfaction of stronger coverage criteria on the state machine. Additionally, this approach provides test oracles in the form of state constraints to test cases derived from sequence diagrams.

We use an automated teller machine (ATM) from an industrial case study to clarify our approach. In this case study, sequence diagrams are used to describe the behavior of the ATM for one customer including, e.g., checks for the correct PIN. The sequence diagrams were used to derive a state machine that contains the behavior of all sequences. Furthermore, the state machine was extended manually and contains additional loops that, e.g., allow the repeated usage of the ATM for several customers. While the sequence diagrams describe the interaction of several objects, the state machine only describes the behavior of the ATM. Behavior of other systems like AR will not be considered in this paper.

This paper is structured as follows. The next section contains the related work. Section 3 comprises the proposed combination of state machine and sequence diagrams. The case study is presented in Section 4. The final section concludes and provides future prospects.

2 Related Work

There has been a lot of work about model-based testing much of which is condensed in [11]. Especially, state machines and sequence diagrams of the Unified Modeling Language (UML) [4] are often used to model test cases. As one example, Nebut et al. [3] derive test cases from contracts such as use cases and sequence diagrams. As another example, Abdurazik and Offutt provide an approach for automatic test generation from state machines [5]. In contrast to that, we aim at the combined use of both diagrams to generate test suites.

There has also been work about the combination of sequence diagrams and state machines. From the very beginning, it was clear that sequence diagrams can describe single transition sequences of a state machine. Bertolino et al. [1] combine both diagrams to derive “reasonably” complete test models to achieve early results for partially modeled systems. Sokenou [9] and Nagy [2] showed furthermore, that the state machine’s start states to execute sequence diagrams can be very important. We extend these approaches by identifying transition sequences instead of start states that are matching to sequence diagrams. Additionally, we combine several sequence diagrams by matching start and end sequences of the
Corresponding transition sequences to build new and more complex sequences, and we propose coverage criteria based on sequence diagrams.

3 Combination of State Machine and Sequence Diagram

Sequence diagrams are often used to describe test cases manually (see e.g. UML Testing Profile and TTCN-3 [7]) where a message to a given lifeline is considered as test input to the corresponding object under test. Due to missing state information, however, the sequences cannot be concatenated. State machines describe a possibly infinite set of test cases but the proper selection of concrete test cases is a complex task. In this section, we describe how to concatenate sequence diagrams by retracing them as transition sequences in state machines and how to take advantage of this concatenation. For that, we denote several states as follows: The initial state is the initial state of the state machine as defined in the UML specification [4, page 521]. Other states are used to refer to the start or the end of an interaction sequence described in a sequence diagram: A start state of a sequence is a state in the state machine that allows to start the execution of the sequence. The corresponding target state of this execution is called end state.

3.1 Advantages of Concatenating Sequence Diagrams

Before describing the concatenation of sequence diagrams, we motivate this approach by listing some of the resulting advantages. First, the set of possible start states of a sequence seq is in many cases disjoint to the set of the state machine's initial states. In order to execute seq, we have to find a sequence seq2 from one of the state machine's initial states to one of seq's start states s. We call such a seq2 an "initialization sequence" for seq if seq2's set of start states contains at least one of the state machine's initial states and s is the end state of one of seq2's transitions. Since there are often several sequences whose possible start states overlap with the state machine's initial states, all these sequences can be used as initialization sequences for other sequences. So, the first advantage of concatenating sequence diagrams is to reuse existing sequence diagrams as initialization sequences for sequence diagrams that cannot be executed from an initial state.

Second, the concatenation of sequence diagrams results in longer and possibly fewer test cases. In some environments like embedded systems, the initialization of test cases results in higher costs than the test execution. Thus, the combination of many test cases into a few long ones can save expenses.

Third, as we will show in our case study, the concatenation of sequence diagrams can result in the detection of faults that are undetected by the execution of single sequence diagrams. Concatenated sequence diagrams can be used to test the concatenated behavior of several sequences as well as their repeatability.

Finally, the presented possibility of sequence diagram concatenation introduces new means of quality measurement for executing sequence diagrams. Until
now, the sole execution of a sequence diagram can be measured, e.g. with the coverage criterion *All-paths sequence diagram coverage* [11, page 122]. Furthermore, the identification of matching (initialization) transition sequences allows the sequential execution of sequence diagrams. We would call the corresponding coverage criterion *All-Context-Sequences*. For the concatenation of sequence diagrams, we can introduce further coverage criteria that are similar to existing transition-based coverage criteria: For instance, *All-Sequence-Pairs* (the concatenation of all pairs of sequence diagrams), *All-n-Sequences* (the concatenation of all n-tuples of sequence diagrams), or *All-Sequence-Paths* (all sequence diagram concatenations of any length). These new coverage criteria might be useful for evaluating the test suites derived from a state machine and a set of sequence diagrams. Their use, however, has to be evaluated in case studies.

### 3.2 Formal Definitions

There are many different kinds of state machine definitions (cf. [8, 10]). In this paper, we stick to the definitions of state machines as presented in the UML 2.1 specification [4]. We sketch basic elements of this definition. A state machine $sm \in SM$ is a set of regions $Reg$ and pseudo states $PS$. Each region contains a set of vertices $Vert$ (i.e. states and pseudo states) and a set of transitions $Trans$. Transitions $trans \in Trans$ connect vertices – they reference a corresponding source and target vertex ($trans.source$, $trans.target$). Each transition contains instances of trigger $T_{sm}$, guard $G$, and effect $E$. The set of all transitions in state machine $Trans_{sm}$ is derived as union of transition sets from all regions. Based on this, we denote $Trans_{Seq}$ as the set of all transition sequences of a state machine. Each transition $trans \in Trans$ has a label $t[\{g\}/\{e\}]$ where $t \in T_{sm}$ is the trigger, $g \in G$ is the guard $trans.guard$ of $trans$, and $e \in E$ is the effect $trans.effect$ of $trans$. Both elements $g$ and $e$ are of type OCL. We only consider deterministic state machines to have an unambiguous mapping from triggers to transitions.

A sequence diagram is defined as a 5-tupel $SD = (L, M, O, \Lambda_{sd}, T_{sd})$. The set $L$ represents the life lines of all objects in the given sequence. The relation $M = L \times \Lambda_{sd} \times L$ includes all messages sent between life lines. Each message $m \in M$ has a label $t(\{para\}) \in \Lambda_{sd}$ where $t \in T_{sd}$ is the trigger of the message and $para$ represents all parameters of $t$. $O = M \times M$ defines the ordering of messages. A message $m_1$ is called before a message $m_2$ in a sequence iff $(m_1, m_2) \in O$. In our scenario, $O$ is total, i.e. there are no asynchronous messages.

Triggers are method calls or events send to an object. A sequence diagram’s set of triggers $T_l \subset T_{sd}$ for incoming messages of a lifeline $l$ is a subset of all triggers $T_{sm}$ of the corresponding state machine $sm \in SM$ if $sm$ is associated to $l$, i.e. describes the behavior of the object of $l$. A message $m \in M$ with the label $t(\{para\})$ can be executed in a given state $s \in Vert$ iff $\exists trans \in Trans : trans.source = s$ and $trans.guard$ is satisfied by the current attribute and parameter value assignment resulting from the already executed messages. Our definition conforms to UML protocol state machines and considers only explicitly modeled transitions.
3.3 Concatenate Sequence Diagrams

This section contains the descriptions of how to concatenate sequence diagrams by concatenating corresponding transition sequences of a state machine. For that, we have to derive state machine transition sequences from sequence diagrams. We introduce the function \( \text{findTransitionSequences} : SD \times SM \rightarrow \mathcal{P}(TS) \) that produces transition sequences for each combination of sequence diagram and state machine. Afterwards, we show how to concatenate the transition sequences.

In [9], we described a method to derive a set of possible start states for the execution of behavior defined in sequence diagrams. In contrast, this paper is focused on deriving and comparing transition sequences instead of single states. Thus, we present an algorithm to derive transition sequences from state machines that reflect the described behavior of sequence diagrams. Fig. 1 shows the algorithm of the corresponding function \( \text{findTransitionSequences} \). Due to reasons of conciseness, the pseudocode leaves out some aspects of transition matching such as transition guards, post conditions or state invariants. Nevertheless, we are aware that tracing the current system state (i.e. system attribute value assignment) is important to determine important aspects of transition matching, e.g. the satisfaction of transition guards.

```plaintext
findTransitionSequences (sequenceDiagram, stateMachine) {
    sequences = empty set; // return value
    msg = first message of sequenceDiagram;
    startStates = all states of stateMachine with outgoing transitions triggered by msg;
    for each(s in startStates) {
        tmpS = s;
        transitionSequence = empty sequence;
        for(i = 0; i < number of messages of sequenceDiagram; ++i) {
            msg = sequenceDiagram.messages[i];
            if(tmpS has outgoing transition t triggered by msg) {
                tmpS = target state of t;
                add t to transitionSequence;
            } else {
                transitionSequence = empty sequence;
                break;
            }
        }
        if(transitionSequence is not empty) {
            add transitionSequence to sequences;
        }
    }
    return sequences;
}
```

Fig. 1. Algorithm for detecting all transition sequences for a sequence diagram.
For each sequence diagram, we retraced its described behavior as a sequence of transitions in the state machine. In this section, we use these transition sequences to combine several sequence diagrams: For two transition sequences \( t_s_1, t_s_2 \in TS \) with \( t_s_1 \) is assumed to be executed before \( t_s_2 \), we consider three cases: 1) \( t_s_1 \) does not include a transition whose target state is the start state of \( t_s_2 \). In this case, both transitions cannot be concatenated. 2) The target state of the last transition in \( t_s_1 \) is equal to the source state of the first transition in \( t_s_2 \) and 3) an end sequence \( t_s_{2E} \) of \( t_s_1 \) is equal to a start sequence \( t_s_{2A} \) of \( t_s_2 \) (see Fig. 2). For the second and the third case, \( t_s_2 \) can be executed after \( t_s_1 \) (without executing the overlapping transitions in \( t_s_{1B} / t_s_{2A} \) twice).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{Overlapping transition sequences.}
\end{figure}

4 Case Study

In this section, we present a scenario in which sequence diagrams were derived from requirements of an automated teller machine (ATM) and applied to create test cases for the ATM. Furthermore, these sequence diagrams are composed to a state machine. Originally, test cases were only derived directly from requirements, not from models. In the following, we show how to use our approach to generate more complex test cases for such scenarios.

The state machine is shown in Fig. 3. Transitions of state machine are numbered so it is easier to describe transition sequences in the following. A possible behavior of a customer that enters his PIN (4 digits) and withdraws money is described for instance by the following transition sequence: \( (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8, 2) \) (cf. Fig. 3). In the figure, \( AC \) stands for Account Check and \( AR \) stands for Authorization System. Note that the state machine describes just the behavior of the ATM. Thus, some messages of the sequence diagrams are not included in the state machine.

All sequence diagrams are directly derived from requirements use cases. Example sequences are shown in Fig. 4, 5 and 6. Sequence 1 is an initializing sequence as it can be executed in the initial state of the state machine, state Idle. The message \( ec \text{ inserted} \) can only be executed in state Idle, thus the algorithm starts with transition \( ec \text{ inserted} \) in the state machine. Following the algorithm in Fig. 1, transition sequences for Sequence 1 are: \( transseq_{1,1} = (4, 14, 15, 15, 15, 24, 23) \) and \( transseq_{1,2} = (4, 14, 15, 15, 15, 15, 24, 21) \). The last transitions of both sequences depend on the values of the attributes.
own_customer and foreign_customer. For Sequence 2, only one transition sequence can be found: transseq₂ = (20, 19, 13, 8). There is also only one transition sequence for Sequence 3: transseq₃ = (8, 2).

The target state of the last transition of transseq₂,1 is the same as the source state of first transition of transseq₂,2. Thus, both are concatenated to build a new scenario and a resulting new transition sequence transseq₂,1,2 = (4, 14, 15, 15, 15, 24, 23, 20, 19, 13, 8). As the transition sequence transseq₂,2 ends with the same transition as transseq₃, also Sequence 3 can be concatenated to the others. One of the resulting transition sequences is transseq₂,1,2,₃ = (4, 14, 15, 15, 15, 24, 23, 20, 19, 13, 8, 2). Corresponding to the transition sequences, we concatenated the sequence diagrams and created longer test cases. Fig. 7 shows the resulting sequence diagram.

As we expected, several faults can be detected by longer sequence diagrams. For instance, the combined sequence is the only sequence in the case study that
Fig. 4. Sequence Diagram 1

Fig. 5. Sequence Diagram 2

Fig. 6. Sequence Diagram 3
contains a scenario from inserting EC card until money and EC removal. Furthermore, we found a sequence that was not covered by the original sequence diagrams but can be derived by combining sequence diagrams: The original sequence diagrams only describe that a customer enters a wrong PIN three times without removing the EC card in between. However, there is no sequence diagram for a customer that three times inserts an EC card, enters the PIN incorrectly just once, and cancels the operation afterwards. With the repetition of several complete scenarios for interactions of customer and ATM, such scenarios can be covered. We found these improvements by manual inspection. It would be interesting to automate this approach to identify and evaluate further advantages.

5 Conclusion and Outlook

In this paper, we presented an approach to combine sequence diagrams by retracing their described behavior as transition sequences in a state machine. We presented a concrete algorithm, listed several advantages of this approach, e.g. mentioned possible resulting coverage criteria, and showed the applicability of our approach for an industrial case study.
Sequence diagrams are used to describe typical scenarios. They often do not define conditions for the initial state of execution and describe only parts of a scenario. Thus, the presented concatenation of sequence diagrams is a good way to create more complex test cases while avoiding to initialize each sequence separately before its execution. This concatenation can be automated and, thus, no additional manual effort is necessary. Additionally, the proposed combination of two diagrams allows to define new coverage criteria based on both diagrams.

There are some points left to discuss. First, combining sequences can result in a lot of new test cases. More complex test cases can reduce test effort but increase the effort to find errors. To reduce complexity, the number of these test cases might be reduced using the proposed coverage criteria. Second, we have to define limitations on the algorithm. It seems to make no sense to combine sequence diagrams that overlap in all transitions except one or to generate only one combined test case from all sequences. What is the maximum overlap that should be taken in consideration? What is the minimum number of test cases?

In the future, we plan to implement the presented approach, e.g., as an extension of the tool ParTeG [6]. We want to use the tool to automatically create test cases for concatenated sequence diagrams. Here, we will take also the proposed new coverage criteria on sequence diagrams into account.

References

Model Based Testing of Web Service Composition Using UML Profile

Irum Rauf1, Muhammad Zohaib Z. Iqbal2, Zafar I. Malik3

1Dept. of Information Technologies, Åbo Akademi University, Turku, Finland.
2Department of Computer Science, International Islamic University, Islamabad, Pakistan
3Ministry of Education, Islamabad, Pakistan
irauf@abo.fi1, zohaib.zafar@iiu.edu.pk2, zafarimalik@hotmail.com3

Abstract.
Composite web services are made by combining existing web services. The existing web services are like a black-box for a composite web service developer and he has little or no knowledge about their internal workings. These services are loosely coupled and can be selected from a repository of the services at run time. The service is deployed on vendor’s machine and is not under the control of the composite web service developer. This might result in a number of compatibility issues among web services when they are integrated with each other. In this paper, we present an approach for testing of web service compositions using UML profile for Business Process Execution Language (BPEL). The UML profile models are used for the model based testing of the composite web service. Information on contracts is extracted from the specification models and both black-box and white-box testing strategies are applied to validate the process of composition and the functionality of a composite web service. A case study of a Travel Arrangement Service shows the applicability of the approach.

Keywords: Web Service, Web Service Composition (WSC), BPEL, UML- Profile

1 Introduction

A web service is software that uses internet protocols and other web-related standards and provides various services to its subscribers. It offers interoperability and accessibility over the internet. The web services can be discovered and used by the consumer as required. Traditional testing techniques are considered inadequate for testing a web service since the services are loosely coupled in a Service Oriented Architecture (SOA). In traditional testing techniques, the functionality of software is known and it does not change its binding at runtime. In the similar manner, component based architecture deploys a component that is tested for the functionality it provides. The components are physically deployed so that even if a new version of the component comes, it does not affect the already deployed component. In SOA, a
service is loosely coupled and it can change its binding at runtime or a vendor can change its service functionality since it is deployed at vendor’s machine. This difference, on one hand is the main benefit of SOA and on the other hand offers a great challenge to the developers and testers of the SOA-based software.

A composite web service needs to make sure that the services participating in the composition process provide the functionality they are intended to provide, the process of a composing the services to form a new composite web service is working as required, and the composite web service provides the functionality according to its specifications. This requires a combination of white-box and black-box testing strategies to ensure that all the validation tasks are performed, thus making the task of testing a composite web services a complex procedure.

In recent years, much R&D effort have been focused on the testing of composite web service, however, the use of models in the testing procedure have not been addressed much. Models precisely define the intended use of a system and also serve as part of specification document. Unified Modeling Language (UML) is established as a de facto standard modeling language. Various efforts have been done to model web service composition in UML. In [19], a survey of various work done in UML based composition of web services is presented. BPEL is considered semantically rich, yet simple and widely accepted at industrial level among different WSC languages. The UML profile for BPEL has been published by IBM in 2003 [3] and Ambühler[11] provides UML 2.0 Profile for WS-BPEL in his thesis along with transformation rules from the UML models to BPEL [11].

Our work uses UML profile for BPEL for the testing of a composite web service. The UML profile models that are developed as specification models are used in the testing process to extract the information of contracts instrumented in BPEL code. It must be noted that we do not need to test whether the implementation is according to specification as our models are executable. Since, in SOA services are loosely coupled and can be selected dynamically from a repository, so our testing process aims to ensure that every service that is deployed at runtime for the composite web service provides the intended behavior, their results do not disrupt the composite process and the resulting functionality of the composite web service. In this line of work, our work provides a novel approach that used different UML Profile models developed during the design phase extracts information from these models to generate contracts for each activity of the BPEL process and generates test cases for the validation of the BPEL process and the resultant composite web service.

2 Related Work

Various techniques have been proposed for the testing of Web Service Composition (WSC). Tsai et al [1] [2] terms testing of web service composition as a collaborative effort between all the involved parties. Their work involves selecting the right web service from various candidate web services for the WSC development. Bertolino and Polini [4] also address the issues of selecting the right candidate service for the composite web service and propose an audition framework that validates a web service before registering it. The reported literature shows that mostly the composite
web services under test are written in BPEL. In [5], [6], [7], [8] and [9], white box testing strategies are used to validate a composite web service.

Zheng et al [5] have used model checking for test case generation. BPEL is converted to formal models in Web Service Automata (WSA). SPIN and NuSMV model checkers are used for test case generation and state, transition, and du-path coverage criteria are used for control flow and data flow testing of BPEL. Test cases are generated to test whether the implementation conforms to the specifications or not. In [6], Dong et al show how High-Level Petri-Nets are constructed from BPEL-based WSC. The existing testing techniques developed for HPN can then be used for the translated HPN for the testing of WSC. This paper provides a detailed description on how BPEL can be translated to HPN but does not focus much on providing a testing technique. Li et al [10] propose a BPEL unit testing framework. This includes BPEL process composition model, test architecture, lifecycle management and test process design.

Yan et al. [7] provide a BPEL test case generation approach based on concurrent path analysis. BPEL is first converted to Extended Control Flow Graph (XCFG) and then sequential paths are generated from it. These sequential paths are combined to concurrent test path. In [8], Yuan et al also provide a test case generation approach that deals with BPEL concurrent semantics. They call their extended control flow graph as BFG. The concurrent test paths are generated by traversing the BFG model and test data is generated using constraint solving methods. Mayer and Lubke [9] discuss a layer-based architecture for white-box BPEL unit-testing. It describes different layers and proposed several implementation techniques.

A detailed study concerning data flow modeling for testing of composite web services is provided by Bartolini et al in [20]. This work provides an overview of the different ways in which data flow modeling and analysis can be used for the validation of WSC and also provides a WSC case study of the Virtual Scientific Book Store to illustrate some of the data flow approaches. A comprehensive survey on recent research work on SOA testing is provided by Canfora et al in [21]. Their survey analyses the challenges from the viewpoints of different stakeholders and solutions for WSC testing at different levels are given that include unit, integration, and regression testing. Both functional and non-functional testing techniques for SOA are explored in this work. Bertolino et al present the PLASTIC validation framework [22], which combines various verification techniques for both functional and extra-functional properties which include both off-line and on-line testing stages. The framework aims to cover the whole service life-cycle addressing all the aspects of testing a service. It is not a fixed methodology but rather considered as a collection or set of different techniques and approaches which can be used alternatively depending on the requirements of each application.

Our analysis shows that BPEL is a widely accepted WSC language. The majority of the work is done using BPEL as input language and verifying the compatibility of web services with each other. However, there is an obvious lack of use of a modeling language that is well accepted and widely used at industrial level, e.g., UML.
3 UML Profile based testing architecture

The purpose of our technique is to validate the functionality of composite web services and that the web-services participating in the composition process are compatible with each other. This needs to be checked because web services are developed independent of the composition process they would be participating in and may be changed by vendor since it is deployed at vendor’s machine. Figure 1 shows the framework for the model based testing of WSC. The approach contains the following modules: Analyzer Module, Generate test case module, Instrumentation module and Execution module.

The behavioral and structural aspects of a BPEL process are represented by various models. These models are required as input to the technique. We must note here that development of these models is not an extra effort on the part of the tester, since these models are developed during the design phase by the developer. In order to generate test paths, the activity diagram of the BPEL process is taken as input to ‘generate test path’ activity. These paths are generated using different coverage criteria. Activity diagram is also used as input to the ‘generate test data’ activity and test data is generated using different black-box testing techniques. Test cases are generated by combining sets of test paths and test data. These test cases are executed on BPEL code instrumented with pre and post conditions.

Using UML BPEL Profile, different aspects of the system are modeled in structural models i.e., BPEL Process, Interface, Protocol and MessageContent models[3], that include information on the structure of the process, the type of links with the partner, operations of an interface and the types of messages respectively.

The behavioral feature of a BPEL process is modeled using an activity diagram in UML BPEL Profile[3]. It show different activities a process participates in to result a composite web service.

3.1 Analyze Module

This module takes as input the UML BPEL profile models representing structural and behavioral aspects of BPEL process. The activity analyzes all the models according to process flow illustrated in Figure 2. We extract the information about attributes and the associated interfaces. We aim to generate a list of pre and post conditions involving these attributes and generate contracts for the associated interfaces. This needs further exploration. We aim to generate this list in Object Constraint Language (OCL). OCL is a formal specification language that describes rules applied to UML. Analysis and discussion on the sub-activities (invoke, assign, receive, and reply) is not included here due to space limitation.

3.2 Generate Test Case Module

Test cases consist of test data and test path. The set of test paths are appended with set of test data as a Cartesian product. This results in test cases that are executed to test the BPEL process. The test paths are generated as explained below.
Generate Test Paths. This sub-activity takes as input the activity model and generates test paths using different coverage criteria. Test paths provide the traversal path in the activity model based on different coverage criteria. We can apply white-box coverage criteria to test the working of the BPEL process, since it is developed by the BPEL process developer and its source code is available. The UML BPEL Profile activity model is taken as input to this sub-activity.

Fig. 1. UML Profile based approach for WSC Testing

In order to traverse the activity diagram, we use the approach of Linzhang et al. [14]. We use following coverage criteria to generate test paths from the activity diagram:

- **Decision coverage criterion [13]:** This coverage criterion is taken from the work of Myers. It states that true and false outcome of each decision should be taken into account at least once.
- **Multiple condition coverage criterion [13]:** This criterion requires that every combination of condition in each decision should be tested and every point of entry should be invoked at least once.
- **All edge coverage criterion [15]:** This coverage criterion suggests that every edge of the activity diagram should be traversed.
- **n-Path coverage criterion [16]:** According to this criterion, a test engineer specifies a number 'n', i.e., the number of test paths that shall be executed.
- **All Activities Coverage Criteria [17]:** This coverage criterion suggests that every activity in the activity diagram should be traversed.
Fig. 2. Process flow for Analyze Activities

**Generate Test Data**
Test data is generated for the test paths generated in the ‘generate test path activity’. Test data is generated only for the first activity as rest of the activities follow the traversal path. Test data is also generated for guard conditions and instrumented in BPEL code [12]. This test data can be generated using various test data generation techniques. We have adopted the following test data generation techniques for our approach from the work of Myers [13]
- Equivalence Partitioning
- Boundary-Value Analysis

### 3.3 Instrumentation and Execution Module
In this module, the developer instruments the code with pre and post conditions generated from ‘Analyzer Module’, and also using the information provided by the specification document. We plan to automate this step in our future work such that the role of developer is reduced and we provide a complete automated web service composition process from the design to the testing phase.
The ‘Execution Module’ includes **Generate Test Oracle** and **Execute Test Cases** activities. We plan to work on this module in our future work. We will be working on generating test oracles using existing techniques in the literature and employing the post conditions of the contracts as test oracles for our test cases.
Table 1. Pre-conditions and Post-conditions of various activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Pre-condition</th>
<th>Post-conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokeDA</td>
<td>flightDetails. Destination &lt;&gt; NULL &amp;&amp; isString(flightDetails.destination)</td>
<td>flightResponseDA.Price&lt;&gt; NULL &amp;&amp; isFloat(flightResponseDA.Price)</td>
</tr>
<tr>
<td></td>
<td>flightDetails.departureDate &lt;&gt; NULL &amp;&amp; isDate(flightDetails.departureDate)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; flightDetails.travelclass &lt;&gt; NULL &amp;&amp; isString(flightDetails.travelclass)</td>
<td></td>
</tr>
<tr>
<td>assignEA</td>
<td>flightResponseEA.Price&lt;&gt; NULL &amp;&amp; isFloat(flightResponseEA.Price)</td>
<td>travelResponse.Price= NULL &amp;&amp; isFloat(travelResponse.Price)</td>
</tr>
<tr>
<td></td>
<td>travelResponse.Price= NULL &amp;&amp; isFloat(travelResponse.Price)</td>
<td></td>
</tr>
</tbody>
</table>

For test execution, we advocate the use of BPEL as a test execution language. BPEL has been used earlier as a test execution language by Li et al. [10]. The execution engine will run test cases on BPEL code instrumented with pre and post-condition, and will compare the execution results with the test oracle. Based on comparison results, we will conclude whether a given test case is pass or fail. These pass and fail results will be employed further for diagnosing location the fault. However, this work is currently out of scope of our work.

4 Case Study

This case study of an Employee Travel Arrangements Service is taken from Juric [18] and has been modified to add new requirements. The service is used by the end-user who is interested in getting a cheaper available ticket for flying to his destination address. The availability of ticket is selected based on his traveling class by which he often travels. The service initiates when the user of the service provides his name, destination, departure date and a return date. The Employee Travel Status Service returns the information about the class by which the user travels. If the user travels by business class, then Emirates Airlines and Gulf Airlines are invoked for the prices. If the user travels by ‘economy’ or ‘first’ class, then American Airlines and Delta Airlines are invoked. The prices obtained from these Web Services are compared and the lower price is selected and returned to the client.

Our testing approach requires input models representing various structural and behavioral aspects of the composition under test. Based on requirements and details of the required composite web service, we have developed input models based on UML Profile of BPEL. Due to space limitation, only the behavioral model (activity model) of the case study is shown in Figure 3.
4.1 Analyzer Module

This activity results in a list of pre- and post-conditions after applying sub-activities for invoke, receive, assign and reply activities explained earlier. An excerpt of the list of pre- and post condition generated by this activity is shown in Table 1. 

*Generate Test Paths:* From the activity diagram shown in Figure 4, we generate test paths that will help us in identifying bugs. We use tabular structure to represent essential information of our activity diagram. This tabular representation has been previously used by Linzhang et al in their work [14]. The test paths are generated using various coverage criteria previously discussed and the test cases are generated by combining test data and test paths.

- *Decision Coverage Criterion:* It generates 4 test paths and 8 test cases

![Activity model for 'Employee Travel Arrangement Service.'](image-url)
• **Multiple Condition Coverage Criterion**: It generates 18 test paths and 36 test cases.
• **All Edge Coverage Criterion**: It generates 4 test paths and 8 test cases.
• **'n' Path Coverage Criterion**: If \( n = 7 \), then any 7 test paths are generated randomly, and 14 test cases.
• **All Activities Coverage Criteria**: It generates 4 test paths and 8 test cases.

Due to space limitation, only the test-paths generated using Decision Coverage Criterion are shown in Table 2. As discussed in section 3, these test-paths are converted to test-cases and are executed to obtain the results. Here, \( TP \) is Test Path, \( A_i \) (where \( i=0,1,2,...,n \)) are the activities and \( T_i \) (where \( i=0,1,2,...,n \)) are the activity edges.

### 5 Conclusions and Future Work

In our work, we have presented a framework for the model-based testing of WSC using UML Profile for BPEL and also highlighted many issues that need to be addressed in providing an automated approach for validating a composite web service. The presented work aims to test the compatibility of web services involved in the composite web services. This is necessary in SOA domain since web service may change over a period of time and the composite web service developer may not be aware of it. This paper provides and overview of the methodology that can be adopted for the model based testing of a composite web services. A case-study of Employee Travel Arrangements Service has also been discussed that shows the applicability of the proposed approach. There are a lot of open issues that can be addressed as future work of our approach, including generating contracts information, implementing strategies of contracts and evaluating the effectiveness of our testing plan. Our work uses both black-box and white-box testing strategy to address various aspects in validating a composite web service.

### References

Engineering and Advanced Applications


SESSION 3
Tool Support for MBT
Case Study-Based Performance Evaluation of Reactive Planning Tester

Andres Kull¹, Kullo Raiend¹, Jüri Vain², Marko Kääramees²

¹ Elvior Ltd, Mustamäe tee 44, 10621 Tallinn, Estonia
{Andres.Kull, Kullo.Raiend}@elvior.ee

² Dept. of Comp. Sci., Tallinn Univ. of Tech., Raja 15, 12618 Tallinn, Estonia
vain@ioc.ee, marko.kaaramees@ttu.ee

Abstract. Contemporary on-the-fly model-based test generators focusing mainly on computationally cheap but far from optimal planning strategies cover just a fraction of the wide spectrum of test control strategies. Typical examples of those used are simple random choice and anti-aot. Exhaustive planning during online testing looks out of reach because of the low scalability of the methods in regard to the model size. The reactive planning tester (RPT) studied in the paper is targeted to fill the gap between these two extremes. The key idea of RPT lies in offline learning of the SUT model to prepare the data for efficient online reactive planning. Based on an industrial scale case study, namely the city lighting system controller, we demonstrate that tuning the planning horizon of RPT allows a trade-off to be found between close to optimal (in terms of test length) and scalability of tester behaviour with computationally feasible expenses.

Keywords: model-based testing, reactive planning, on-the-fly test generation, nondeterministic model.

1 Introduction

On-the-fly test generation is widely considered to be the most appropriate technique for nondeterministic implementation under test (IUT) [1], [2]. The state-space explosion experienced in many offline test generation methods is eliminated by on-the-fly techniques because only a limited part of the state-space needs to be kept track of at any point in time. On the other hand, exhaustive planning is difficult on the fly due to the limitations of the available computational resources to meet the required response time. Random choice is the simplest way of selecting the next test stimuli. This is inefficient because it is based on the random exploration of the state space and leads to test cases that are unreasonably long and nevertheless may leave the test goal unachieved.

The on-the-fly test generation method implemented as a reactive planning tester (RPT) [3] is a trade-off between using a simple heuristic and exhaustive planning. The principles of reactive planning are applied to the problem of test planning under uncertainty. Reactive planning operates in a timely fashion and hence can cope with highly dynamic and unpredictable environments [4]. Just one subsequent input is
computed at every step based on the current context. Instead of producing a complete
test plan with branches (test tree), a set of decision rules is produced. The rules are
constructed by applying offline analysis on the given finite state model of the IUT and
the test goal. From the state model of the IUT the RPT and the decision rules are
synthesised. The RPT is able to generate test inputs on the fly depending on the
observed reactions of the IUT and the test goal. It selects the next move by calculating
the gain function for the possible moves and selects the one with the highest gain [3].

Although the RPT can plan and select its next move efficiently it still has to
calculate the gain functions online. Depending on the state model size the gain
functions can become quite complex and make the online calculations time-
consuming. In this paper we introduce the control of the planning horizon that can be
used in the RPT synthesis phase. The control can be used to experiment with the
planning horizon and select the most suitable one for each test case.

In [3] some experiments on a small model were conducted to compare RPT
efficiency against the random choice and anti-ant [10] tests selection methods.
Experiments with larger models were not possible because of the lack of tool support.
By the time of writing the paper the MOTES test generator [5] for RPT synthesis was
available making experiments with bigger models possible. The paper presents the
industry-scale case study and the test generation measurement results to affirm the
statements about the RPT performance against the random choice and anti-ant test
selection methods. The results of the experiments give guidance for selecting the most
suitable planning horizon as a compromise between planning efficiency and gain
functions calculation time.

2 Reactive Planning Tester

2.1 Synthesis of Reactive Planning Tester

A detailed description about the synthesis of RPT is given in [3]. The tester model is
constructed as a dual automaton of the IUT model where the inputs and outputs are
inverted. A test goal is a specific objective or a property of the IUT that the tester is
set out to test. We focus on test goals that can be defined as a set of Boolean traps
associated with the transitions of the IUT model. The goal of the tester is to generate a
test sequence so that all traps are visited at least once during the test.

The tester makes its choice in the current state based on the structure of the tester
model and the bindings of the trap variables representing the test goal. The gain
functions and gain guard [3] are constructed for transitions of the tester model
controllable by RPT. The RPT synthesiser analyses the structure of the IUT model
offline and learns possibly successful IUT runs regarding the test goal. The gain guard
evaluates true or false at the time of the execution of the tester determining if the
transition can be taken from the current state or not. The value true means that
taking the transition is the best possible choice to reach some unvisited traps from the
current state. Gain functions are used to define gain guards. Gain functions calculate
the gain that is a quantitative measure needed to compare alternative choices of test stimuli on the fly. For each controllable by tester transition a non-negative gain function is defined that depends on the current bindings of the context variables. The gain guard of the tester controllable transition is true only when the transition is a prefix of the test sequence with highest gain among those that depart the current state. If several gain functions evaluate to the same maximum value the tester selects one of the best transitions optionally randomly or by the “least visited first” principle. Each transition on the model is considered to have unit weight and the cost of testing is proportional to the length of the test sequence. The gain function of a transition computes a value that depends on the distance-weighted reachability of the unvisited traps from the given transition.

For deriving the gain functions the reduced shortest-paths tree (RSPT) on IUT dual automaton is constructed for each controllable transition. The root vertex of the tree corresponds to the controllable transition being characterised with gain function, other vertices present transitions equipped with traps. In case there are branches without traps in the tree that terminate with terminals labelled with traps the branches are substituted with hyper-edges having weights equal to the length of that branch. By the given construction the RSPT represents the shortest paths from the root transition it characterises to all reachable trap labelled transitions in the tester model. The gain function also allocates weights to the trap labels in the tree, and closer to the root transition the higher weight is given to the trap. Thus, the gain value decreases during the test execution after each trap in the tree gets visited.

2.2 Bounded Planning Horizon

Since the gain functions are constructed based on RSPTs their complexity is in direct correlation with the size of RSPT. In that way, the all transitions coverage criterion sets the number of traps equal to the number of transitions in the IUT model. Considering the fact that the number of transitions in the full-scale IUT model may reach hundreds or even more, the gain functions generated using RSPTs may grow over a size feasible to compute at test execution time. To keep the on-line computation time within acceptable limits RSPT pruning is added to the RPT synthesis technique. The planning horizon defines the depth of the RSPT to be pruned. Although the pruning of RSPT makes online planning incomplete it makes the RPT method fully scalable regardless of the size of IUT model and the test goal. Moreover, there is an option to set the planning horizon automatically offline when specifying the upper size limit to the RSPT pruned. Pruning of RSPT reduces the resolution capability of RPT gain functions. To resolve the priority conflicts between transitions having equal maximum gain values RPT uses either random or anti-ant choice mechanisms. Both conflict resolution approaches are discussed in more detail in the next section.
3 Reactive Planning Tester vs. Other On-the-Fly Testing Methods

3.1 Random Choice

Random selection with uniform distribution of enabled transitions (random choice) is the simplest method for selecting the next move from the possible moves in nondeterministic systems’ on-the-fly testing. In random choice no test sequence has an advantage over the others and the concrete one is taken randomly. The planning horizon of the random choice is zero steps ahead. Random choice is a very fast method of selecting the next move but it is inefficient because it is based on the random exploration of the state space and leads to test cases that are unreasonably long and nevertheless may leave the test goal unachieved.

Random choice has been used in the early TorX tool [6], Uppaal-Tron [7], [8] and also in the on-the-fly testing mode of SpecExplorer [9].

3.2 Anti-Ant

The anti-ant method tries to avoid already visited paths. For selecting the next move from the possible ones the anti-ant method takes the less visited one. The planning horizon of the anti-ant algorithm is 1 step ahead in case of all transitions coverage.

The anti-ant heuristic-based state space exploration method is introduced in [10] and used in [11] to cover all transitions of the labelled transition system resulting in the exploration of a model program.

3.3 RPT

Compared to random choice with a planning horizon 0 steps ahead and anti-ant with 1 step ahead the planning horizon of RPT can be significantly longer because the method “sees” the direction of the next remaining traps. Because of the longer planning horizon RPT can result in shorter test sequences than random choice and anti-ant methods. The payback of the shorter test sequence is the complexity of the gain functions needed to be calculated online. The advantages of RPT over the pure anti-ant and random choice methods depend on many factors like the model size and the complexity, the degree of nondeterminism of the IUT, and the distribution of the traps over the model.

Model size and complexity affect the size and complexity of the RSPT, and therefore also the size and complexity of the gain functions.

The degree of the nondeterminism of the IUT is a factor that can make the better planning worthless. The planning may target to reach a trap behind some nondeterministic choice point in the model but due to the nondeterminism the reachability of the coverage item is not granted. If there are many nondeterministic choices in the model and on the path to the next trap then the nondeterminism can direct the test execution to the paths other than to be preferred by gain guards.
The distribution of traps in the model may affect the resulting test sequence length by different methods. If the traps are in deep loops then random choice and anti-ant may not be able to reach them at all or it may take an unreasonably long time because those methods do not “see” where the traps are located. By offline analysis the RPT knows where the traps are and tries purposefully to reach them. Eventually it does even if there are nondeterministic choices on the path. Of course, the IUT as a nondeterministic transition system must be fair for the transitions to be fully covered, which means that all transitions from the nondeterministic choice points are eventually possible. If there is a relatively small amount of traps distributed in the model then again random choice and anti-ant are less effective than RPT because the RPT can “see” where the unvisited traps are located and can plan the path towards them. Random-choice and anti-ant can also reach them using a significantly longer test sequence. If there are lot of traps in the model (for example, all transitions test coverage criterion) then anti-ant performs relatively better than random choice.

4 Experiments

The experiments are made to prove the feasibility of the RPT method and to compare its performance with the random choice and anti-ant methods using an industry scale case study.

4.1 The Case Study

The testing case study developed under the ITEA2 D-MINT project [12] evaluates the model-based testing technology in the telematics domain. The IUT of the case study is a Feeder Box Control Unit (FBCU) of the street lighting subsystem.

The most important functionality of the FBCU is to control street lighting lamps either locally, depending on the local light sensor and calendar, or remotely from the monitoring centre. In addition, the controller monitors the feeder box alarms and performs power consumption measurements. The communication between the controller and monitoring centre is implemented using GSM communication.

The RPT performance evaluation experiments are performed on the powering up procedure of the FBCU.

4.2 Test Environment

The test environment (Fig. 1) consists of the FBCU as the IUT, Elvier MessageMagic TTCN-3 test development and execution platform [13], and a system adapter that transforms FBCU electrical output signals to MessageMagic input messages and MessageMagic output messages to electrical input signals of FBCU. The MOTES test generator [5] generates TTCN-3 tests for MessageMagic out of the IUT models prepared using Gentleware’s Poseidon for UML CASE tool [14].
4.3 Model of the IUT

The model implements the power-up scenario of the FBCU. The strongly connected state model of the FBCU includes 31 states and 78 transitions. The model is nondeterministic. Pairs of nondeterministic transitions depart from seven states of the model and a triple of nondeterministic transitions departs from one state of the model. The minimum length of the sequences of transitions from the initial state to the farthest transition is 20 transitions, i.e. the largest depth of the RSPT for any transition is 20.

The model is similar to the model of code lock with several nested loops. There are several possibilities to fall from the successful scenario back to the first states if something goes wrong in the scenario.

4.4 Planning of Experiments

In order to demonstrate the algorithms in different test generation conditions we varied the test coverage criterion. The tests were generated using two different coverage criteria – all transitions and a single selected transition. The single transition was selected to be the farthest one from the initial state. The location of the single transition was selected on the limit of the maximum planning horizon. Different RPT planning horizons (0 to 20 steps) were used in the experiments. In case the RPT planning resulted in several equally good subsequent transitions in the experiment with the selected coverage criterion and planning horizon we used alternatively the anti-ant and random choice methods for choosing the next transition. If the planning horizon is zero then RPT works like the pure random choice or anti-ant method depending on the option selected in the experiment.

As a characteristic of scalability we measured the length of test sequences and time spent online on each planning step. The planning time is indicative partially only because it depends on the performance of the RPT executing platform. Still, those measurements give some hints about the scalability of the method with respect to the planning horizon. In addition to the nondeterministic model there is always a random
component involved in the RPT planning method. Therefore we performed all experiments in series of 30 measurements and calculated averages and standard deviations over the series.

4.5 Results and Interpretation of the Experiments

The experiments are summarised in Table 1. The lengths of the test sequences are given in the form \( \text{average} \pm \text{standard deviation} \) of 30 experiments. The results in the first row of Table 1 with planning horizon 0 correspond to the results of the pure anti-ant and random choice methods.

For estimation of the minimum test sequence length we modified the examined nondeterministic model to the corresponding deterministic model with the same structure. Eliminating the nondeterminism in the model by introducing mutually exclusive transition guards and using the maximum planning horizon 20 the reactive planning tester generated the test sequence with length 207 for all transitions coverage criteria on the modified deterministic model. The minimum length of the test sequence to reach the single selected transition was 20 steps.

Table 1. Average lengths of the test sequences in the experiments.

<table>
<thead>
<tr>
<th>Horizon</th>
<th>All transitions test coverage</th>
<th>Single transition test coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Planning with horizon</td>
<td>Planning with horizon</td>
</tr>
<tr>
<td></td>
<td>anti-ant</td>
<td>random choice</td>
</tr>
<tr>
<td>0</td>
<td>18345 ± 5311</td>
<td>44595 ± 19550</td>
</tr>
<tr>
<td>1</td>
<td>18417 ± 4003</td>
<td>19725 ± 7017</td>
</tr>
<tr>
<td>2</td>
<td>5120 ± 1678</td>
<td>4935 ± 1875</td>
</tr>
<tr>
<td>3</td>
<td>4187 ± 978</td>
<td>3610 ± 2338</td>
</tr>
<tr>
<td>4</td>
<td>2504 ± 815</td>
<td>2077 ± 552</td>
</tr>
<tr>
<td>5</td>
<td>2261 ± 612</td>
<td>1276 ± 426</td>
</tr>
<tr>
<td>6</td>
<td>2288 ± 491</td>
<td>1172 ± 387</td>
</tr>
<tr>
<td>7</td>
<td>1374 ± 346</td>
<td>762 ± 177</td>
</tr>
<tr>
<td>8</td>
<td>851 ± 304</td>
<td>548 ± 165</td>
</tr>
<tr>
<td>9</td>
<td>701 ± 240</td>
<td>395 ± 86</td>
</tr>
<tr>
<td>10</td>
<td>406 ± 102</td>
<td>329 ± 57</td>
</tr>
<tr>
<td>11</td>
<td>337 ± 72</td>
<td>311 ± 58</td>
</tr>
<tr>
<td>12</td>
<td>323 ± 61</td>
<td>284 ± 38</td>
</tr>
<tr>
<td>13</td>
<td>326 ± 64</td>
<td>298 ± 44</td>
</tr>
<tr>
<td>14</td>
<td>335 ± 64</td>
<td>295 ± 40</td>
</tr>
<tr>
<td>15</td>
<td>324 ± 59</td>
<td>295 ± 42</td>
</tr>
<tr>
<td>16</td>
<td>332 ± 51</td>
<td>291 ± 52</td>
</tr>
<tr>
<td>17</td>
<td>324 ± 59</td>
<td>284 ± 32</td>
</tr>
<tr>
<td>18</td>
<td>326 ± 66</td>
<td>307 ± 47</td>
</tr>
<tr>
<td>19</td>
<td>319 ± 55</td>
<td>287 ± 29</td>
</tr>
<tr>
<td>20</td>
<td>319 ± 68</td>
<td>305 ± 43</td>
</tr>
</tbody>
</table>
The experiment shows that the reactive planning tester with maximum planning horizon results on average in a test sequence many times shorter and a considerably lower standard deviation than the anti-ant and random choice tester. For the test goal to cover all transitions of the nondeterministic model the RPT generated an average test sequence 1.5 times longer than the minimum possible sequence. The difference from the optimum is mainly due to the nondeterminism of the model. Compared to the RPT with the maximum planning horizon the anti-ant and random choice tester generated test sequences that were on average 57 and 146 times longer, respectively.

If the test goal is to cover one selected transition the RPT reached the goal with the length of test sequence that is close to optimal. The anti-ant and random choice tester required on average 104 and 235 times longer test sequences. This experiment shows that the anti-ant tester outperforms the random choice tester by more than twice on average with smaller standard deviation. This confirms the results reported in [11].

The dependency of the test sequence length on the planning horizon is shown in Fig. 2. Non-smoothness of the curves is caused by the relatively small number of experiments and large standard deviation of the results. The planning horizon can be reduced to half of the maximum planning horizon without significant loss of average test sequence lengths for all transitions coverage criterion in this model. Even if planning few steps ahead significantly shorter test sequences were obtained than in case of the random or anti-ant methods. For instance, when the planning horizon is restricted to 2 or 5 steps the average test sequence length decreases by approximately 4 or 8 times, respectively, compared to the anti-ant and random methods. If the test goal is to cover a single transition then the test sequence length decreases exponentially to the value of planning horizon.

![Graph showing test sequence lengths for different planning horizons](image)

Fig. 2. Average test sequence lengths of the test sequences satisfying the all transitions (left) and single transition (right) test goal.

At planning horizons less than maximum effects of priority conflict resolution methods do not have clear preference. The anti-ant method performs better for all horizon lengths in case of the single transition coverage criterion (Fig. 2, right) and for small values of horizon length in case of all transitions coverage (Fig. 2, left). The random choice method performs better on average for horizon lengths from 4 to 10 (Fig. 2, left) for this model for the all transitions coverage criterion.

We also measured time spent for planning one step. The average duration of a planning step in milliseconds is shown in Fig. 3. The computer used for experiments 94
has an Intel Core 2 Duo E6600 processor running at 2.4 GHz. Experiments on the
model demonstrate that the growth of planning time with respect to the planning
horizon is not more than quadratic. The average time for calculating the gain function
values with a maximum planning horizon in one step is less than 9 milliseconds.
When the planning horizon is increased to maximum then the average depth of the
shortest paths trees remains below the maximum horizon and the average planning
time stabilises.

![Graph showing average time spent for online planning of the next step.](image)

**Fig. 3.** Average time spent for online planning of the next step.

## 5 Conclusions

We have studied the performance of the online reactive planning tester in comparison
with the random choice and anti-ant methods. Based on an industrial scale case study,
namely the city lighting system controller, we demonstrated that tuning the planning
horizon of RPT allows us to reach close to optimal (in terms of test length) tester
behaviour with computationally feasible expenses on the strongly connected state
model with 31 states and 78 transitions.

The experiments were performed using two different coverage criteria – all
transitions and a single selected transition. Different RPT planning horizons (0 to 20
steps) were used in the experiments. All experiments were performed using both the
anti-ant and random choice selection of the subsequent transition in case the RPT
planning resulted in several equally good next transitions.

Experiments on the case study model show that by increasing the planning horizon
the average length of the test sequences decreases exponentially while the planning
time of the next step increases not more than by a power of 2 in the planning horizon.
In the experiments the planning time remained within the range of 13-22 ms.

The feasibility of RPT for online testing is demonstrated in the practical industry
scale case study. RPT significantly outperforms the anti-ant and random choice
methods. On average the RPT with a maximum planning horizon resulted in test
sequences length less than a hundredth that of the pure anti-ant and random choice.

Generalisation of the results of performance analyses requires additional
experiments with different case studies in the future.
Acknowledgments. This work was partially supported by the Estonian Science Foundation under grant No. 7667, by the ELIKO Competence Centre project “System Validation and Testing” and by the ITEA2 D-MINT project.

References

Compliance Test Framework

L. Bouwmeester, G. Broadfoot and P. Hopcroft

Verum, Laan van Diepenvoorde 32, 5582LA, Waalre, The Netherlands
Email {Leon.Bouwmeester, Guy.Broadfoot, Philippa.Hopcroft}@verum.com

Abstract. This paper presents the Compliance Test Framework (CTF), a proof of concept tool chain that was developed within an industrial setting for the purposes of providing a rigorous, statistically-based testing environment for large and complex embedded software systems. CTF is successfully in use by Philips Healthcare to certify major sub-systems before they are integrated with other sub-systems and this paper describes the application of this approach that was done for a Cardio-Vascular X-ray system.

1 Introduction

It has become increasing common for products of all types to contain complex embedded software to control devices and machines. Defects in such software, therefore, increasingly cause product failure and can have important safety consequences for the users. The software the authors work with in practice falls into this category and typically has the following characteristics: Firstly, it is complex and very large, ranging in size from tens of thousand to tens of millions of source lines. Common examples include a modern wafer stepper that is controlled by 12 million source lines of control software and a modern cardiovascular X-Ray system that is controlled by 17 million source lines of control software. Secondly, the software is event driven, reactive and must remain responsive to external events over which it has no control. Thirdly, such software is highly concurrent, since it must be able to control many actions and processes in parallel. Fourthly, such software is business critical in the sense that software failures can result in severe economic loss for a business. Finally, in all cases, the externally observable functional behavior is nondeterministic and therefore conventional testing, no matter how extensive, cannot provide meaningful measures of correctness.

It is widely recognized that existing testing practices applied to such software suffer from the following drawbacks: 1) Too few test cases for results to be statistically meaningful and no means of determining the amount of testing needed, 2) economic infeasibility of generating sufficiently large sets of test cases to be statistically meaningful, 3) a high proportion of erroneous test cases being produced, 4) no guarantee that the sample has been taken from a complete population, 5) the possibility of using invalid models as the testing environment, 6) limited confidence levels and 7) limited or no traceability between the informal specifications describing the test environment, the system-under-test and the results of failures reported during the execution of the test cases. As a result, it is common for testing to require a
substantial part of the software development cycle and budget, while only yielding a small degree of confidence regarding the reliability of the software.

An alternative approach to conventional testing was proposed by [5,6,10] in which statistically significant test sets are generated from a representation of system behavior called a usage model. The results of the test set execution are used to derive statistical measures of system reliability i.e., the likelihood of errors occurring in the long term use of the system. The validity of the results depends, inter alia, on the correctness and completeness of the usage model. An incomplete usage model implies that the population from which the test sequences are drawn is not complete and therefore some behavior will never be tested irrespective of how many tests sets are generated. An incorrect usage model will result in invalid test cases being generated and will give false results. In the work of Poore et al usage model validation is performed by manual inspection but this approach is not feasible for industrial scale systems. The industrial application of these techniques therefore requires some means of automatically validating the usage model.

This paper describes the authors’ solution to validating usage models and their experience of applying this approach in practice to industrial scale systems by extending the existing usage modeling concepts [11,12] and JUMBL [6], and integrating them into a practical tool chain. An example of this approach is its use for testing a Cardio-Vascular X-ray system within Philips Healthcare, an overview of which is also presented.

2 Background

This section gives an overview of the statistical testing and usage modeling concepts, as well as the ASD technology upon which the remainder of this paper is dependent. For clarity, the software being tested is referred to as the system-under-test or SUT.

2.1 Statistical testing and usage models

A statistical approach to software testing was proposed by Harlan Mills [4] and developed further by [5,6,10] and is based on similar scientific approaches to product testing and certification in other engineering disciplines. Here, products are typically certified under protocols in which random samples of the product are drawn, test characteristics of operational use are applied, analytical or statistical inferences are made and products that meet a specified standard are then certified as fit for use.

To apply the principle of statistical testing, a model of software use, known as a usage model, is developed, test cases are generated randomly from the usage model and test results then allow reasoning about the entire population of use in terms of software quality and test sufficiency. Software use can be modeled as a finite state, time homogeneous Markov chain [12] and the standard analytical results for Markov chains can be interpreted to yield insight about long term operational use. The structure of a usage model represents the possible use of the software; a probability distribution is then imposed on this structure representing expected use of the software. Although this approach is based on black box testing and therefore does not
prove that the SUT is error-free, it does result in higher quality products by both objective and subjective assessments.

The most common format for describing usage models is The Model Language (TML) [7], a simple language developed for specifying Markov chain usage models. TML allows specifying the probabilities as constraints along with simple objective functions to simplify the specification of the probability distribution of a usage model.

The process of statistical testing based on a usage model is as follows: 1) One or more usage models are constructed to represent the population of uses; 2) usage models are manually analyzed and compared against known and/or assumed properties of field use and test constraints; 3) test cases are generated and test automation is planned and designed; 4) test cases are executed against the SUT and results are recorded; and 5) the results of the tests are analyzed to determine expected reliability.

Steps 1 and 2 are performed manually; steps 3 and 5 are automated by JUMBL, the J Usage Model Builder Library [6]. JUMBL generates two types of test sets: The first one is the minimum coverage test set, which is a collection of test cases covering all arcs of the usage model. This serves as a “smoke test” and establishes whether or not the SUT is of sufficient quality for statistical testing to be meaningful. The second type is the random test set, which randomly chooses outgoing arcs in a model according the given probabilities.

2.2 Analytical Software Design (ASD)

Analytical Software Design is model-based component technology supported by a toolset for constructing industrial scale systems from verified components. ASD provides fully automated formal specification and design verification as well as automatic code generation for C++, C# and C. ASD guarantees equivalence between ASD models of interfaces and designs, the corresponding formal models and the runtime behavior of the generated code. ASD models are based on the Sequence-based Specification (SBS) method [8,9] to specify both interfaces and designs, thereby ensuring completeness of the ASD models in the sense that in every state, the consequences of all stimuli are specified; if a stimulus is not allowed or expected in a given state, then it is explicitly specified as illegal. These specifications are traceable to the original requirements and remain completely accessible to critical stakeholders. This allows them to play a key role in validating the ASD models and retain control over them. At the same time, ASD models provide the rigor and precision necessary for the formal verification. The SBS method [8,9] was extended to support nondeterminism for the purposes of specifying interface behavior.

The ASD toolset supports fully automatic generation of formal models from ASD models. The formal models are based on the process algebra CSP [2] and verification is done using the FDR model checker [3]. Model checking is used to verify whether a design satisfies its functional requirements, is free from race conditions with its used interfaces and free from deadlocks and livelocks. The compositional property of CSP refinement provides the necessary scalability for industrial size systems.
3 Case: Client-Server Architecture

This section presents an overview of the architecture of a Cardio-Vascular X-ray system comprising 3 major sub-systems: 1) A Front-End comprising a patient table and C-arc, which contains a/o the X-ray generator and the image detector producing raw images, 2) an Image-Processor that takes the raw images and processes them into clinical images (which is left out from the figure below for the purposes of this paper) and 3) a Back-End, where the clinical images can be viewed, stored, post-processed, and where the patient scheduling and examination selection is done.

![Formal model of the behavior of IFeBe that is visible to the BE component.](image)

Fig. 1. Cardio-Vascular X-ray System

The interface between the Front-End and the Back-End sub-systems (IFeBe) has been formally specified using ASD, resulting in a precise, complete and correct interface description. If the design of the sub-systems were specified with ASD, then the design could have been formally verified against the interface. However, due to the presence of legacy code this is not possible and raises the question of how to determine whether the existing software behaves according the formal interface specification.

The interface between the Front-End and the Back-End is used as a case for the approach presented in this paper. It contains synchronous stimuli (representing procedure calls initiated by the Back-End), corresponding synchronous return values and asynchronous call-backs that are decoupled by a FIFO queue. Figure 1 shows that the Back-End acts a client towards the Front-End server. The Back-End is abstracted from the Front-End by using an ASD model of the interface between Back-End and Front-End. Given that the Back-End contains legacy code it is not possible to formally verify that this Back-End is using the interface in a correct way. Instead, the Compliance Test Framework will be used to make statements on software reliability.

4 Compliance Testing Framework

The Compliance Testing Framework (CTF) enables software engineers to specify a usage model in the ASD input language, verify its correctness with respect to its used interfaces of the components with which it interacts in the actual system and automatically generate test cases according to a statistical measure. This framework also provides a test environment within which the test cases are executed in practice,
4.1 Extending the usage models

The ASD tool set is extended to support usage modeling: usage models are described in terms of sequence-based specifications thereby ensuring completeness, where all legal and illegal behavior is explicitly specified. This enables the usage models to be formally verified with respect to its used interfaces to ensure that it represents a complete and correct population from which the test cases are sampled. The ASD tool set supports automatic generation of TML from these usage models which is then automatically fed into JUMBL to generate coverage tests or random test sets after which compile C++ test cases are automatically generated. Usage models as described in ASD allow the use of Boolean expressions called predicates, which effectively introduces history in the usage model. As usage models are based on first order Markov chains, these predicates are automatically eliminated by generating equivalent states and thereby ensuring that the usage models remain statistically correct. Furthermore, the ASD tool set also automatically insert calls to test interfaces wherever specified in the usage models.

Section 3 presented the case of the Back-End of a cardio-vascular X-ray system, where the question of how to determine whether the Back-End uses the interface correctly was raised. Using the principles of statistical testing this implies the creation of a usage model of the Back-End, describing it use. Test cases are generated from this usage model and used to sample the behavior of the Back-End (by testing) to determine the probability that the Back-End behaves in a manner sufficiently indistinguishable to the usage model according to some statistical measure.

4.2 Test case execution

Figure 2 shows the (simplified) interaction of a test case, sampled and generated from the usage model by JUMBL, with the SUT.

In addition to the test case, SUT and the used interface are the following elements:

Fig. 2. Interaction test case with SUT
• Test interface: This tests reaction to error conditions, it is necessary to inject errors in the SUT. Furthermore, through this interface the SUT will be reset to a proper state in case a test case will fail to ensure that it is in a known state for the execution of the next test case. Finally, the test interface is also used to trigger actions in the SUT; e.g. simulating pressing buttons on a user interface.

• Adaptor: This makes the test environment indistinguishable from the actual environment. It implements the IFeBe interface as if it was a true Front-End. The observable behavior during testing defines what can and cannot be stated about a compliant SUT. Because the inner workings of the SUT are hidden, compliance testing cannot distinguish incorrect internal behavior unless this behavior results in incorrect observable behavior during testing. An “incorrect SUT” may therefore still be compliant. Compliance Testing cannot “prove” that a SUT is compliant under all possible circumstances. It can only state that:

1. The executed set of tests is constructed such that it is statistically meaningful.
2. When executing the tests, all observed behavior is compliant.
3. The probability that noncompliant behavior remains undetected is within the specified confidence and reliability parameters.

However, these conclusions are only as valid as the usage model from which the test cases are sampled. A usage model that is incomplete or contains invalid sequences of behavior renders the conclusions drawn from the statistical analysis meaningless in practice. It is therefore necessary to establish a set of properties that a usage model must satisfy, as presented in the next section.

5 Verification of usage models

As described above, the effectiveness of a usage model and how meaningful the sample of test sequences are, is dependent upon whether the usage model is “correct” according to a suitable correctness criteria. This section presents 3 correctness properties that a usage model must satisfy, known as compliance, validity and completeness. Each property is defined in turn below using the architectural assumptions of the case example presented in Section 3, with a single used interface of the Front End. In particular, the client-server architecture is assumed, where the Back-End is the SUT and the client, the Front-End behaves according to the ASD interface specification IFeBe and is the server and the asynchronous call-backs from Front-End to Back-End are decoupled by a non-blocking FIFO queue.

Furthermore, for a given usage model UM and used interface IFeBe, the complete model System is defined as the parallel composition of UM, IFeBe and a FIFO queue, synchronizing upon all common communication shared, with the exception of the asynchronous call-backs that go via the queue.

5.1 Compliance

The first correctness property is one that is automatically verified for all standard ASD designs and their corresponding used interfaces and is referred to as compliancy.
of a design model with respect to its used interfaces. This definition is extended to incorporate usage models as follows.

A usage model UM is defined to be compliant with respect to an interface IFeBe precisely when the complete model, System, satisfies the following conditions:

1. No illegal behavior is invoked in either UM or IFeBe. As described in Section 3, a usage model is defined in ASD as a complete sequence based specification. Therefore, any behavior that is deemed illegal is explicitly specified and modeled as such. Similarly for used interface models. It is then straightforward using the standard ASD refinement checks in FDR to verify whether either component can cause the other to behave in a way that is specified as illegal.

2. For a given queue length N, the queue is non-blocking in System. This is an assumption made in the actual implementation and must therefore be reflected in the corresponding model.

3. System is deadlock free.

4. System is livelock free: The components cannot engage in an infinite sequence of internal chatter to the detriment of the other.

These basic compliance properties are expected to be satisfied by any well behaving SUT. Therefore, the usage model reflecting the SUT must also satisfy them. All properties above are easily verified automatically within the ASD toolset as part of the CTF tool chain. Any failures of compliance are directly traceable to the model specifications and can therefore be analyzed and corrected by the domain experts of the SUT.

5.2 Validity

A usage model that is proven to be compliant with respect to its used interfaces may still contain invalid test sequences. An invalid test sequence is one that does not represent a possible execution path required by the actual SUT. Proving compliance does not prove that there is not a set of unwanted or incorrect behavior in the usage model that was not invoked by the well-behaving used interfaces. Therefore, the second correctness property states that every test sequence in the usage model must be a valid execution that the actual SUT is required to perform with respect to its used interfaces. If this was not the case, then invalid test sequences could lead to false negative results. This is not acceptable in practice, as it causes substantial resources, in terms of manpower and costly system setup, to be devoted to looking for errors that may or may not be in the actual SUT.

A usage model UM is defined to be valid with respect to an interface IFeBe precisely when all legal sequences of behavior specified in the UM model are in the set of sequences that UM can perform when constrained in the complete parallel composition System. This property is easily and automatically verified within the ASD toolset using CSP refinement and the FDR model checker.

UM will not satisfy this property in the case where there exists a legal sequence of behavior in UM that is not possible when UM is running in parallel with IFeBe and its queue. This would reflect the fact that there is additional behavior in the usage model that is not invoked or handled by its used interfaces. This does not necessarily imply that the additional behavior is erroneous or non-reflective of the actual SUT. It may
also be the case that the SUT does indeed have more behavior that is not invoked by its actual used components. However, the premise is that the formal used interface specifications are a correct description of the interfaces of the used components in the actual system and it is therefore not desirable to generate test sequences that these components could not perform in practice. Additional “unused” implemented behavior of the SUT is regarded as being outside of the scope of the population definition of test cases.

5.3 Completeness

The third correctness property relates to ensuring that the population from which the test sequences are generated is complete with respect to the used interfaces. In other words, the usage model must be able to handle all possible communication that well-behaving used components can do with the SUT in the actual system.

A usage model UM is defined to be complete with respect to an interface IFeBe precisely when all legal sequences of behavior specified in the IFeBe model are in the set of sequences that IFeBe can perform when constrained in the complete parallel composition System. This property is easily and automatically verified within the ASD toolset using CSP refinement and the FDR model checker.

This does not imply that all legal sequences in UM are also in the IFeBe interface or that the UM will generate all test sequences that are available from the IFeBe’s point of view, since their communication is decoupled by the queue. Rather, it implies that the UM does not constrain the possible legal behavior of IFeBe and therefore has sufficient behavior in its specification to handle the complete range of IFeBe’s behavior. If the UM is unable to co-operate with one or more sequences that is specified as legal in IFeBe then UM would fail this completeness requirement. Given that the formal used interfaces are assumed to accurately model the complete communication between the actual used components and the SUT, it is essential that the test sequences are drawn from a population that reflects this.

6 Results in Practice

A proof of concept tool chain has been implemented and applied in practice. A user defines a usage model based on ASD models and verifies that it is compliant, valid and complete according to the definitions presented above. This usage model is automatically transformed into a first order Markov model in TML from which JUMBL creates either a coverage test set or a random test set. The generated test sequences are then translated into compilable C++ test cases. During this translation, the tool automatically inserts calls to a logger component to keep track of all steps performed in a test case. This ensures that the statistics can be calculated properly afterwards and also provides a complete trace to the point of failure for failed test cases. Once the usage model is specified, almost all subsequent steps in the tool chain are fully automated.

This proof of concept tool chain is being used by Philips Healthcare, business unit Cardio-Vascular. Firstly, they made an ASD interface specification between the
Back-End and the Front-End. The verification of this interface model revealed several millions of execution scenarios, leading Philips Healthcare to conclude that it was infeasible to test such components using existing practices. Consequently, they decided to incorporate CTF in their development process. Currently, a usage model has been implemented for the Back-End sub-system and been verified for compliance, validity and correctness; currently, this usage model contains about 1600 rule cases. From this usage model, test cases have been generated sampling the Back-End. These test cases typically contained several thousands of steps confirming the infeasibility to test such complex systems manually. These test cases are used to sample the Back-End in order to determine whether the required reliability level has been met, in which case it will be certified and can be integrated with other certified sub-systems. Execution of these test cases revealed several errors in the Back-End system. For example, an error was revealed that showed a race condition between exchanging patient data on one hand and image acquisition on the other. The interface protocol between Back-End and Front-End specified that exchange of patient data and image acquisition had to be mutually exclusive to ensure consistency between Back-End and Front-End. This work is being extended to also test the Front-End implementations.

7 Conclusions

This paper presented an overview of the Compliance Test Framework, a proof of concept tool chain that was developed within an industrial setting for the purposes of providing a rigorous, statistically-based testing environment for large and complex embedded software systems. CTF has been implemented and is now successfully in use by Philips Healthcare to certify major sub-systems before they are integrated with other sub-systems. Given the size of the test cases, it demonstrated how infeasible existing practices of manually generating test cases and an accurate test environment are for such complex systems and at the same time it also demonstrated the feasibility and practicality of the technology as presented in this paper.

During the development of CTF, a number of challenges and new research avenues were raised that are currently being pursued. Firstly, the initial version of CTF generated compiled C++ test cases. In practice, the number of test cases and the sizes of individual test cases sometimes lead to difficulties in compiling and linking these test cases. Current research is being done into interpretive test case execution instead of generating them as source programs. Secondly, the test boundary from which the test cases are generated must be the same as the actual test boundary that the test cases are being executed with respect to the SUT. Currently, the test cases are generated from the usage model, which is defined upon the output side of the queue. This is the logical test boundary for defining usage models and one that would be preferable to avoid increasing the complexity of already large models. In practice, however, it is difficult to run the test environment behind the callback queue; it is more common for such queues to be integrated into the SUT itself. Current work is being done to look at how these differing test boundaries can be reconciled.

Finally, since the testing is done upon visible behavior only of the SUT with respect to a given interface (i.e. black box testing) and does not include details of the
internal implementation, it is common to have a significant amount of nondeterminism arising with respect to responses generated. This can arise due to the decision of which response will occur as a result of a particular input sequence being determined internally and due to the presence of responses that can be generated spontaneously, both of which are beyond the testing environment’s control. Given a test sequence and no ability to control the SUT’s internal implementation, a valid sequence could produce a false negative result, because the system chose a different response sequence that was equally valid behavior, but not the one specified in the test case. In the initial CTF implementation this was handled partly by implementing a test interface that provided a higher degree of control of the SUT and partly by restricting certain sets of test cases. Current research is being done to develop an alternative approach for handling responses from the SUT at runtime and for determining the success or failure of a given test case.

Acknowledgements

We are grateful to Philips Healthcare for allowing us to present this case study and for their support when we applied these techniques to the Back-End sub-system of the Cardio-Vascular X-Ray system.

References


106
Emerging Patterns for Testing Model Management Tools

Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Tara Gilliam and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
[louis,dkolovos,paige,rg,fiona]@cs.york.ac.uk

Abstract. The quality of model management tools, such as model transformation and validation tools, is of paramount importance to the adoption and success of Model-Driven Engineering (MDE). To build confidence in the operational correctness of such model management tools and avoid regressions, establishing a rigorous and automated testing process is crucial. In general, model management tools consume models, produce models or both. As such, to test such a tool in a black-box manner, one needs to be able to provide test input models, and check that the operation produces the expected results. This paper contributes to the current state of practice by presenting and comparing a set of novel approaches for performing these two tasks based on our experience of testing the languages and tools of the Epsilon platform.

1 Introduction

In the past, studies [3, 12] have found that the evolution of software can account for as much as 90% of a development budget; there is no reason to believe that the situation is different now. Sjøberg [19] identifies reasons for software evolution, which include addressing changing requirements, adapting to new technologies, and architectural restructuring.

Software evolution can cause a system to become resistant to further evolution: “Over time the code will be modified, and the integrity of the system, its integrity according to its design, gradually fades. The code slowly sinks from engineering to hacking.” [5, pg xvi]. A system that is resistant to change, such as the one described above, is termed brittle rather than malleable.

Automated testing is a key to developing malleable software. The ability to regularly run tests affords developers the confidence to improve the structure of existing code [5]. Moreover, when correcting defects a test case can be used to recreate undesired behaviour, and then to guard against regression. Test-driven development can be used to engineer new features alongside tests, and to produce well-structured code [1].

Model-Driven Engineering (MDE) introduces additional challenges for controlling and managing evolution [11] and testing, particularly because of the application of automated model management operations that manipulate and change models. This paper contributes to improved understanding of testing automated operations in the context of MDE.
1.1 Model Management Tools

Model management is the discipline of managing artefacts used in MDE. Model management is typically implemented as a set of model management operations, implemented using languages and tools such as ATL [7], OpenArchitectureWare [16], and Epsilon [9]. Typical model management operations (such as model-to-model transformation, model validation, model merging and code generation) consume models, produce models or both. These operations are designed to be automated and reusable: thus, establishing a rigorous and automated testing process to build confidence on their operational correctness and avoid regressions is critical.

When testing model management operations and tools, we require structures and processes for constructing models and for checking models. In the course of implementing many operations and tools in the Epsilon platform, we have found that existing software testing frameworks do not provide built-in mechanisms for constructing and checking models and, consequently, for testing these tools in a systematic and automated way. This paper contributes a synthesised analysis of our experience of attempting to test model management tools, and in particular presents a set of emerging patterns that we have applied for this testing process. The patterns that we have applied have been developed so as to help to produce model management tests that are malleable and readable, and enhance automation of the testing process.

In Section 2, we introduce relevant terminology and briefly present an existing software testing framework, JUnit. In Section 3, we present model management testing patterns for constructing and for checking models and provide an example implementation for each pattern. Finally, we discuss related work and conclude in Sections 4 and 5.

2 Background and Terminology

A test checks the behaviour (according to some specification) of, or exercises, some aspect of the system-under-test. A test comprises many test cases. Test cases can be specified using assertions on the system-under-test. A test can also include a test fixture, which specifies the context of the test. A test fixture is constructed before test cases are executed and can be shared between test cases.

In this paper, we concentrate on unit testing. Units are the smallest testable parts of the system-under-test. A unit test exercises a single unit. xUnit is the name given to a family of unit-testing frameworks. Implementations exist for several programming languages, such as cUnit for C and dotUnit for .NET. In our work, we use JUnit for Java.

JUnit tests are Java classes, containing methods annotated to identify them as either test cases, fixture set up methods or fixture tear down methods. JUnit provides several methods for specifying assertions, such as assertEquals(Object expected, Object actual) and assertTrue(boolean actual).

Listing 1.1 shows a JUnit test case (annotated with @Test), singletonShouldContainOneItem, which uses an assertion to test the state of the collection
object under test (line 9). A test fixture is used to constructs the collection object under test (lines 3–4). Here, we use the @BeforeClass annotation to define a test fixture that can be shared between test cases.

```java
@BeforeClass
public static void setUp() {
    singleton = new ArrayList<String>();
    singleton.add("itemA");
}
```

```java
@Test
public void singletonShouldContainOneItem() {
    assertEquals(1, singleton.size());
}
```

Listing 1.1. An excerpt of a unit test specified in JUnit.

3 Patterns

In this section, we present the emerging patterns that we have applied while testing model management tools. Each emerging pattern simplifies the construction or checking of models, areas in which we have found existing software testing frameworks to be lacking. For each pattern, we present an example use, along with a brief discussion of motivation and intent. We describe the patterns as emerging because, at this time, their definition is less formal than other patterns (e.g., [6]). A more rigorous definition will emerge through repeated application of these patterns.

For consistency, we will use the families metamodel shown in Figure 1 throughout this section.

![Exemplar families metamodel](image)

Fig. 1. Exemplar families metamodel.

3.1 Constructing Models

Models for a test fixture, like all other models, can be constructed using the tools provided by a modelling framework, such as the Eclipse Modelling Framework...
EMF, for example, can be used to serialise models to XMI and XML documents, which can be stored alongside, or embedded in, unit tests.

Use a metamodel-independent syntax The format in which a modelling framework stores its models may or may not be suitable for specifying test fixtures. For example, XML is optimised for consumption by machines, not humans, and consequently, specifying test fixtures with XML detracts from readability.

Furthermore, the format in which a modelling framework stores its models may use a metamodel-specific concrete syntax. However, in this case information from the metamodel is required to load and store models. Should the metamodel change, the modelling framework may be unable to load existing models. For example, EMF stores all attribute values as strings without type information. Thus, as a result of metamodel evolution, data can be lost when values are inadvertently cast to the wrong type.

To address these issues, we have used a human-readable, metamodel-independent syntax to construct models in test fixtures.

Example: OMG’s Human-Usable Textual Notation (HUTN) defines a metamodel-independent syntax, which aims to conform to human-usability criteria. Below, we use the HUTN implementation described in [17].

```plaintext
Family {
    name: "The Smiths"
    members: Person { name: "Jill" }
}
```

Listing 1.2. HUTN for an exemplar family.

```xml
<?xml version="1.0" encoding="ASCII"?>
<families:Family xmlns:families="families" name="The Smiths">
    <members name="Jill"/>
</families:Family>
```

Listing 1.3. EMF XMI for the same family (XMI metadata omitted for brevity).

As Listings 1.2 and 1.3 clearly demonstrate, when specified in HUTN rather than in XMI, models become easier to read and maintain manually by humans. Furthermore, HUTN provides several shorthand constructs that further enhance its readability and conciseness [14]. Also, Epsilon HUTN reports a greater range of model and metamodel inconsistencies than EMF.

Create a Factory class Metamodel-independent syntaxes such as XMI and HUTN are generally less concise than metamodel-specific syntaxes. For some categories of models, a metamodel-independent syntax can be cumbersome. For example, we have found HUTN to be excessively verbose when instantiating
metaclasses with few features, and when constructing deeply nested model structures. An alternative approach is to define a factory class [6], which comprises one or more factory methods for each metaclass.

Example EMF can be used to generate, from a metamodel, Java code for constructing instances of that metamodel. We define factory classes as singletons [6] decorating the metamodel code generated by EMF. Each factory method can provide parameters for specifying the values of features. For specifying the values of many-typed features, we use a Java 5 vararg parameter (see lines 4 and 7 of Listing 1.4). Consequently, our test fixtures need not construct arrays or lists.

```java
import java.util.Arrays;
import families.*;

public abstract class FamiliesFixtureFactory {
    public static Family createFamily(Dog d, Person... members) {
        final Family f = FamiliesFactory.eINSTANCE.createFamily();
        f.setDog(d);
        f.getMembers().addAll(Arrays.asList(members));
        return f;
    }

    // Omitted for brevity: createDog(), createPerson(String name)
}

// In the test fixture
createFamily(createDog(), createPerson("Jill"));

// Equivalent HUTN
Model {
    contents: Family {
        members: Person {
            name: "Jill" }
        Dog "D" {
    }
}

Listing 1.4. A Factory for constructing test fixtures.
```

Again, test fixtures are more readable when models are specified using factory classes rather than XMI. Factory classes will also highlight metamodel and model inconsistencies, because any breaking changes to the metamodel will produce compilation errors. We have found factory classes to be more concise than the equivalent HUTN when instantiating metaclasses with few features, and when constructing deeply nested model elements.

Anonymous subclass with an initialiser Factories become harder to use when they contain several methods with similar signatures. Furthermore, for each combination of features that are to be set, a new factory method must be
defined. In these cases, we usually use HUTN to define test fixtures. However, this is sometimes undesirable (for example, when constructing deeply nested model elements), and another approach is required.

We considered adding factory methods with a map (between feature names and values) parameter. Factory Girl [4] for Ruby uses exactly this approach. In Java, however, constructing map objects is less readable than in Ruby. Rather than use map objects, we create an anonymous subclass with a non-static initialiser, a technique described by Norvig in [13]. This rarely-used language feature is denoted simply by a pair of curly braces in the body of the class and can contain arbitrary code; the statements are executed on object creation after superclass initialisation, but before the subclass constructor.

**Example:** Below, we construct an anonymous subclass of FamilyImpl, a class generated by EMF from our metamodel. Lines 2 to 4 set the name, address and district of the family within a non-static initialiser. Values for any feature can be set, and in any order. As anonymous classes have no constructor, this is the last initialisation step and will override any previous values. Any non-private, non-final member variable or method is visible in the block.

```java
new FamilyImpl() {
  setName("John");
  setAddress("10 Main Street");
  setDistrict("City of York");
};
```

*Listing 1.5. An exemplar subclass with a non-static initialiser.*

We have found this approach greatly reduces repetition across unit tests when used in conjunction with a default superclass. Below, we define a DefaultFamily class that sets the name attribute to Bill and the address attribute to 10 Main Street. In the test fixture (lines 10 to 12), we override the DefaultFamily with an anonymous subclass that sets the name attribute to John. The value for the address attribute is inherited from DefaultFamily.

```java
// In the factory
public static class DefaultFamily extends FamilyImpl {
  public DefaultFamily() {
    setName("Bill");
    setAddress("10 Main Street");
  }
}

// In the test fixture
new FamiliesFixtureFactory.DefaultFamily() {
  setName("John"); // 10 Main Street remains as address
};
```

*Listing 1.6. Specifying default values in a factory.*
Subclasses with non-static initialisers reduce the number of factory methods required to construct test fixtures. Together with default classes, the technique can be used to reduce repetition in test fixtures.

### 3.2 Checking Models

Often, modelling frameworks provide facilities for comparing models. Depending on the modelling framework, comparison may include superfluous data, such as unique identifiers used only by the modelling framework.

**Use a comparison algorithm** Some modelling frameworks allow developers to customise the way in which models are compared. When specifying unit tests for model management tools, we have found it necessary to define different comparison algorithms for different metamodels.

**Example** We have used the Epsilon Comparison Language (ECL) [8] to define comparison algorithms for our metamodels. ECL is a declarative language. Rules are used to specify comparisons between pairs of model elements. ECL supports sophisticated string matching techniques such as fuzzy and dictionary-based matching. Our test cases use an assertion method, `assertEclEqual(File ecl, Model expected, Model actual)`, which decorates the ECL execution engine.

In Listing 1.7, we specify two rules for matching expected and actual model elements. Two persons are equal when their names are equal (lines 8–12). Two families are equal when their names are equal and their members are equal (lines 1–6). We use ECL’s `matches` operation to compare two collections (line 5).

```java
rule Families
  match f : Expected!Family with g : Actual!Family {
    compare: f.name = g.name and f.members.matches(g.members)
  }

rule Person
  match p : Expected!Person with q : Actual!Person {
    compare: p.name = q.name
  }
```

**Listing 1.7.** An ECL comparison algorithm for the families metamodel.

Only the model elements and values relevant to model comparison need be considered when using a comparison algorithm to check models. Irrelevant data, and data specific to the modelling framework, is ignored.
Use a model inspection language  It is common to define related unit tests, each testing a different set of parameters (for example, when testing boundary conditions). The unit under test may produce very similar output for each of the related unit tests. Some test cases can be re-used across all related unit tests.

Unit tests exercise only one unit of the system-under-test. Our model management tools most often construct models by using several units together. Thus, we have rarely needed, when specifying a unit test, to use a comparison algorithm on complete models. Rather, we have needed to compare fragments of models.

When re-using test cases or comparing fragments of models, we have found using a language tailored to inspecting (navigating and querying) models, such as the Object Constraint Language (OCL) [15], to be effective. Assertions can be re-used across unit tests, and any number of, rather than all, model elements can be inspected and checked.

Example: We have used the Epsilon Object Language (EOL) [10] to check models in our unit tests. Like OCL, EOL provides model navigation constructs and, unlike OCL, constructs for updating models, statement sequencing, and simultaneous access to two or more models.

To simplify the use of EOL to perform unit testing, we have defined a class, ModelWithEolAssertions, that decorates the EOL execution engine and provides methods specific to unit-testing. In particular, ModelWithEolAssertions adds a constructor that allows fragments of EMF models to be loaded, utility methods that allow EOL variables and operations to be specified, and assertion methods that allow checks against the model to be expressed using EOL.

```java
public class FamilyTest {
    private static ModelWithEolAssertions model;

    @BeforeClass
    public static void setup() {
        // exercise unit under test
        model = new ModelWithEolAssertions(...);
        model.setVariable("f", "Family.all.first()");
    }

    @Test
    public void familyShouldContainMemberCalledJohn() {
        model.assertTrue("f.members.exists(p | p.name = 'John')");
    }
}
```

Listing 1.8. An exemplar unit test that uses ModelWithEolAssertions.
In Listing 1.8, the unit-under-test is exercised, and the resulting model fragment is used to construct a new instance of \texttt{ModelWithEolAssertions} (line 6). The \texttt{setVariable} method is used to name a Family in the resulting model fragment. A test case checks that the Family contains a member called John (lines 10-13). EOL provides declarative operations for collections, such as the \texttt{exists} for expressing existential quantification (line 12).

Using a model inspection language allows unit tests to check a subset of, rather than all, model elements and values. Re-use can be achieved between unit tests by adding a common superclass that includes re-usable assertion methods.

4 Related Work

The testing patterns presented in this paper provide a foundation for implementing others. For example, Schuh and Punke \cite{18} describe the ObjectMother pattern for structuring test fixtures. An ObjectMother is a factory that returns named fixtures. For example, the “The Smiths,” a family comprising Jack, Jill and their two children, or “The Does,” a family comprising mother Jane and daughter Alice. ObjectMother’s named fixtures give rise to personae, which provide additional vocabulary for specification of tests. By extending the factory class pattern presented in Section 3.1, ObjectMother could be adapted for testing in the context of MDE.

Lin et al. \cite{21} were the first to suggest using model comparison for testing model transformations. They outline a conceptual framework for structuring and executing model-to-model transformations, which uses model comparison to produce test results. However, \cite{21} does not acknowledge the need for metamodel-specific comparison algorithms, and the drawbacks of using comparison for checking output models (discussed in Section 3.2).

5 Conclusion and Future Work

We have presented, compared and contrasted a set of patterns for constructing and checking models. We have discussed the way in which we have used the patterns to develop unit tests for model management tools.

Discovering and documenting further patterns is ongoing work. We will also make improvements to testing tools to better support the patterns presented here. In particular, the majority of the patterns presented would benefit from support for embedding domain-specific languages in JUnit. Tool support for embedding EOL, ECL and HUTN would allow errors in the specification of unit tests to be reported at compile-time, rather than at run-time.

Acknowledgement. The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2006-2009).
References
