Introducing Model-Based Testing in Industrial Context – An Experience Report

Hartmut Lackner¹, Jaroslav Svacina¹, Stephan Weißleder¹, Mirko Aigner², and Marina Kresse²

¹ Fraunhofer Institute FIRST, Department Embedded Systems, Kekuléstraße 7, 12489 Berlin, Germany
{hartmut.lackner,jaroslav.svacina,stephan.weissleder}@first.fraunhofer.de
http://www.first.fraunhofer.de
² Thales Rail Signalling Solutions GmbH, Colditzstraße 34-36, 12099 Berlin, Germany
{mirko.aigner,marina.kresse}@thalesgroup.com
http://www.thalesgroup.com

Abstract. Model-based testing is an important quality measurement technique. There are several theoretical advantages of model-based testing and experience reports to support them. Model-based testing, however, is not applicable “out-of-the-box”. Each environment requires specific adaptations. Thus, there are serious acceptance thresholds in industry. In this paper, we present a report on our efforts to introduce model-based testing as a testing technique in an industrial context.

1 Introduction

Testing is one of the most important system validation techniques. In model-based testing (MBT), the system under test (SUT) is compared to a system specification in the form of a model. Several languages are used to create system models. We focus on UML state machines. A common approach to model-based testing is to generate a test suite based on the system model, to execute the test suite, and to compare the observed behavior of the SUT to the expected one.

Although model-based testing has a high potential for reducing test costs and increasing test quality, this technique is adopted slowly in industrial practice. In our opinion, the major reason for this is that model-based testing is not applicable “out-of-the-box”, but requires training and adaptation. This results in costs, e.g. for learning modeling languages, for using model-based test generators, and for integrating model-based testing into the existing testing process. In this paper, we as Fraunhofer FIRST engineers report on our efforts to introduce model-based testing as a new testing technique to Thales Rail Signalling Solutions GmbH during a pilot project.

The paper is structured as follows. In the following section, we present the initial project situation. In Section 3, we present the used toolchains. We describe the course of the cooperation in Section 4 (adaptation and formalization of the system model) and Section 5 (implementation of the test adapter). We evaluate
the used test generation approaches in Section 6. In Section 7, we summarize our results and experiences. Finally, we present related work in Section 8 and conclude in Section 9.

2 Initial Project Situation

In this section, we describe the initial situation of our pilot project. In the project, we focused on testing components of the European Train Control System (ETCS). ETCS is a stationary signaling and train protection system, which is developed as part of the European Rail Traffic Management System (ERTMS). The functionality of the ETCS software components are safety-critical need to be certified (see EN 50128 [1]). Thus, significant effort is applied on quality measurement methods like verification, validation, and test.

According to the regulations resulting from the EN 50128 norm for safety-critical systems, the development process of Thales consists of systematic requirements engineering, functional and design specification, implementation, static analysis methods, and different levels of software testing.

The engineers at Thales use different types of models to specify critical parts of the system: The structure of the system is modeled using class diagrams and the behavior is described using state machines. The models are not used for automatic code generation and several parts of the models are described in an informal way, e.g. using pseudocode and prose. The intention of creating these models was to provide an intuitive semi-formal description of the system behavior and to allow for a common understanding of critical system parts.

At the start of the project, we decided to apply MBT for conformance testing of the system models and the implemented components. The system models were already present but they were not used for code generation. Thus, we chose to reuse them as test models instead of creating new test models from scratch.

3 MBT Toolchains

For automatically generating test suites with MBT, we used one industrial and one academic test tool. In this section, we present the two corresponding toolchains that integrate these tools in the test generation process. Figure 1 depicts both toolchains in an activity diagram: The main element is the system model as the input for both toolchains – the left part shows the industrial toolchain, and the right part shows the academic toolchain.

Both toolchains use the same system model as input and generate test code that is compatible to the test adapter provided by Thales engineers. The following two subsections describe both toolchains in more detail.

3.1 Commercial Toolchain

The commercial toolchain uses two tools and a text transformation program based on Prolog: We used Borland Together [2] for formalizing and concretizing the existing system model. Afterwards, we imported the formalized model to
TestDesigner [3] from Leirios (the company name has been changed to Smartest) and generated abstract test cases. TestDesigner creates test cases in the form of XML documents. We used the Prolog transformation program to transform these XML files to CppUnit [4] tests.

3.2 Research Tool ParTeG

As an alternative to using the industrial toolchain, we also used and adapted the free model-based test generation tool ParTeG [5], which is based on the Eclipse Modeling Framework [6]. The input models for ParTeG are UML state machines in the context of UML classes that are both modeled using the UML 2.1 plugins [7]. Possible output formats are JUnit 3.8 and 4.3 [8].

We had access to the sources of ParTeG. Thus, we see the advantages of using ParTeG in the possibility of adapting the necessary test output format and in implementing unexpected features and interpretations of the system model. An anticipated disadvantage of using ParTeG was its prototype-related immaturity.

4 Adaptation and Formalization of the Model

In this section, we describe the necessary model adaptations and formalizations to automate the test case creation. The original system models were provided by the engineers at Thales. The formalization using OCL/UML was done in cooperation with the engineers at the Fraunhofer FIRST based on the requirements. The system models consist of four UML state machines, which describe the communication behavior of several train modules. There is one main machine that
references the other three. These referenced state machines are subcontrollers that describe the failure handling of the train modules. All of these system models have several flaws that are caused by insufficient formalization. In the following, we describe these flaws and how we removed them.

4.1 Formalization

Here, we present the individual steps to formalize the given models.

**Removing Syntactical Errors.** The first thing we discovered is a violation of the UML syntax: Outgoing transitions of the system model’s initial states contain triggers although the UML specifies that such transitions must not have triggers. We solved this violation by transforming the initial state into a state named *Initializing* and creating a new initial state that is connected to the state *Initializing* via an additional transition.

After this transformation, the model was syntactically correct. However, state information is used in the test oracle and the SUT has no state called *Initializing*. Thus, the model does not represent the SUT’s behavior anymore, and every generated test case would fail. As a solution, we removed all test steps that check for the state *Initializing*.

**Observing Finalization Behavior in Subcontrollers.** Observing the successful termination of a subcontroller and returning a corresponding verdict is important for test generation. Thus, it was necessary for us to observe states whose outgoing transitions lead to a subcontroller’s final state. We call the states with the outgoing transitions of interest *Finalize* states. The original models contained untriggered completion transitions to model these state changes. Consequently, the test adapter was not able to trigger the entry of the final state.

As a solution, we added a trigger *notification* to the completion transitions. The result of this is that the test adapter could explicitly trigger these transitions to reach the final state. Since this new trigger is not part of the SUT’s behavior, this solution also needs adaptation of the test adapter.

**Flattening Hierarchical State Machines.** One effect of the previous model transformation is that outgoing transitions of a subcontroller may lead to leaving the subcontroller while a *Finalize* state is active. This behavior is not intended by the Thales engineers. Instead, the SUT has to finalize and terminate the subcontroller after a *Finalize* state has been reached. A model transformation for compensating for this unintended effect consists of creating explicit outgoing transitions for all states of the subcontroller but the *Finalize* states. This corresponds to flattening a part of the system model. Since this introduces additional elements in the model, it increases the coverage of the SUT [9].
Formalizing Conditions. The guard expressions in the model contain no typed variables or constants. Instead, expressions are written in an informal style like prose. We derived a list of all used identifiers (variables and constants). The engineers at Thales provided the corresponding C++ data types and initial values for them. With this type information, we added formalized expressions to the system model using the Object Constraint Language [10].

4.2 Adding A Context

Based on the formal identifiers of the previous formalization step, we created a structural context for the state machine. This context consists of a class diagram and an optional object diagram. TestDesigner needs both of them. For ParTeG, providing the class diagram for the state machine is sufficient for test generation.

In the context class, we defined all formal identifiers as class attributes. In most cases, the mapping of simple C++ data types into the class diagram was straightforward. As an exception, we had to map unsigned integers to the UML data type integer and constrain it in a later step to non-negative values. The context class also contains setter methods for changing values of class attributes and operations to map triggers from the state machine to the test adapter.

The object diagram represents an instance of the context class. For automatic test generation with TestDesigner, the object diagram defines the initial system attribute value assignment.

4.3 Toolchain-Specific Model Transformations

In this subsection, we present toolchain-specific model transformations that were used to overcome restrictions of modeling tools and to keep the compatibility of test cases and the test adapter that is already in use at Thales.

Disjoint Triggers. First, we experienced problems with transitions that contain two or more triggers. In contrast to the UML standard, Together is not able to create transitions with two or more triggers. We solved this problem by splitting the transition into two or more parallel transitions, each handling a single trigger.

This transformation preserves the semantics, but changes the structure. This has an impact on the generated test suite. For instance, the satisfaction of All-Transitions on a system model with split transitions forces the test generator to traverse more transitions and, thus, to create larger test suites. Likewise, this also has an impact on the fault detection capability of the generated test suite [9].

Timed Triggers. Prior to our project, the engineers at Thales established an interface for the test driver to control system time. Due to restrictions of the test generation tools, we use function call events instead of using standard UML time events. A unique name scheme enables the test adapter to map these function calls to time information of the SUT.
Output Format Transformation. The Thales test framework, for which we had to generate test cases, requires test cases to be written in CppUnit. Test-Designer, as a general purpose test case generator, generates XML files but no CppUnit test files. Thus, we provided a text transformation program based on Prolog to convert the XML files into CppUnit files. After this transformation, the test cases from TestDesigner are executable in the test framework.

We integrated CppUnit test code generation for the Thales test framework directly into ParTeG. Thus, the test suite created by ParTeG did not need any further transformation.

5 Test Adaptation

The following section describes the used test adapter. We used the system model for automatic test generation. Since the system model is comparatively close to the implementation, the gap between the abstraction level of the model and the implementation is likewise small. Nevertheless, adaptation is required to execute the abstract test cases. There are several approaches to implement this adaptation, such as the concretion of the test cases by a model-to-text transformation or the use of an additional test adapter that maps from system models to SUT. We used a mixed approach [11, page 285] to bridge the gap.

The corresponding test adapter defines a test interface that is used to execute the partly transformed test cases. It transforms abstract trigger information of the test cases into concrete events and function calls that are forwarded to the controller within the test framework. Furthermore, information about the states of the system model are not explicitly present in the implementation and the test adapter maps them to system attributes in order to check state invariants of the system model. In general, we simplified complex data and represented them in an abstract way in the system model according to the recommendations for building test models by Utting and Legeard [11]. The task of the test adapter was to reinsert this complexity in the test cases.

6 Evaluation

In this section, we evaluate the different approaches to test case creation by comparing the code coverage and the size of the corresponding test suites. Line and branch coverage of the tests are demanded by the certification authorities. Other measures for a test suite’s quality are mentioned in Section 8.

In Table 1, we describe four different test suites: the manually created test suite, the test suite generated by TestDesigner to satisfy All-Transitions, and two test suites generated by ParTeG. The first ParTeG test suite (ParTeG 1) just satisfies Multiple Condition Coverage on the system model, whereas the second one (ParTeG 2) additionally satisfies Multi-Dimensional [12] and contains sneak path analysis and model transformations like flattening the model or splitting choice pseudostates [9].
<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
<th>Number of Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually Created</td>
<td>92.98%</td>
<td>92.86%</td>
<td>26</td>
</tr>
<tr>
<td>TestDesigner</td>
<td>87.19%</td>
<td>83.33%</td>
<td>141</td>
</tr>
<tr>
<td>ParTeG 1</td>
<td>91.32%</td>
<td>91.67%</td>
<td>252</td>
</tr>
<tr>
<td>ParTeG 2</td>
<td>94.63%</td>
<td>92.86%</td>
<td>2280</td>
</tr>
</tbody>
</table>

Table 1. Code coverage and size of the test suites.

Both test generation tools were restricted to use the given system models for test generation. In contrast, the human testers were able to use the models and also additional information like the source code of the controller or abstract information about the system environment like railroad track information. Correspondingly, the achieved coverage of the manually created test suite is higher than most of the automatically generated test suites. The only exception is the test suite for ParTeG 2: It is several times larger than the manually created test suite but covers a higher percentage of the source code. Since ParTeG 2 was generated automatically from the system model, the costs for test generation of this comparatively large test suite are neglectable.

None of the existing test suites covered 100% of lines or branches. The major reason for this is that some of the required test information, such as a model of a railroad track, are not included in the system model and, thus, could not be used for testing.

Since the SUT is already in use, the main objective of our project was not to detect undetected failures, but to improve the existing test process using MBT technologies. Thus, we compared the failure detection capabilities of the test suites using code coverage. Reasons for detected differences were found in the models and requirements.

7 Results and Lessons Learned

In this section, we present the results and lessons learned during the pilot project.

Before the start of the project, the engineers at Thales used models as pictures to support system development. Using these pictures, system designers could communicate their designs to the company’s developers. Both knew about the informal style of the models and communicated directly with each other when something was unclear. Since the designers do not have to take care of precise syntax and semantics, this type of imprecise modeling is easier than designing formal models. For automatic test generation, however, precise and formal models are needed. As presented in the following, creating these formal models for automatic test generation caused more problems than expected.

First, as the applied transformations show, the designers of the system models interpreted the UML in a different way than the test tools do. This kind of semantic error was much harder to fix than the syntax errors. The reason for this is that removing semantic errors needed a detailed investigation and a higher
cooperation effort. Most time was spent on understanding why the design is wrong and how to correct it.

Second, some of the generated test cases are not reasonable. This was caused by missing environment information in the system models. We created no model of the environment and the test adapter did not check for a corresponding consistency of the test data. A solution to this problem is to provide a model of the environment to the test case generators, e.g. by adding information about railroad tracks like turnouts or the current train position.

Furthermore, we consider the repeatability of our actions. The concrete actions for removing syntactic and semantic issues cannot be reused in other projects or on other models because they differ from case to case. For instance, guidelines for designing models may vary for each project. The automatic transformations for adapting the test cases to the test adapter, however, can be repeated. Some transformations (see [9]) are applied to models and can be performed automatically. Transformations like the presented test design tool- and test adapter-specific ones can also be automatically reused in other projects.

Part of our evaluation was also the comparison of manually created test suites and automatically created ones. The large number of test cases generated by ParTeG tests the SUT extensively. One drawback is the execution time of some of the automatically generated test suites. ParTeG 2 designed roughly a hundred times more test cases than the human test designer, resulting in an increased execution time. However, the larger ones of the automatically generated test suites also covered a higher percentage of the SUT than the manually created test suite, and the test design is done automatically.

We also compared the two applied test generation tools. ParTeG generated the test cases in less than ten seconds. TestDesigner needed 25 minutes to generate a test suite. ParTeG reached at least the same or even a higher code coverage than the manual test cases when the strongest generation criteria (ParTeG 2) are applied. In general, test generation is undecidable and each applied test generation algorithm fits only to certain kinds of models. Thus, this is no general comparison of both tools but only an additional measurement of our project.

In retrospective, we encountered many unforeseen obstacles. Although we knew that MBT is not applicable “out-of-the-box” and we were prepared to customize our toolchains, we were surprised by the number of issues. Using even the latest version of the tools did help reducing costs, e.g. for creating the additional test adapter. On the other side, the use of automatic test design also helped saving costs. In contrast to one person week for manually updating the test suites, automatic test generation requires only a few minutes for updating the model and generating the test suite again.

8 Related Work

Several books provide surveys of conventional testing [13–15] and model-based testing [11, 16]. Many modeling languages have been used to create system models. The UML [17] is a popular representative that has been used by many authors
to demonstrate test generation techniques [18, 19]. In this paper, we used UML state machines.

Complete testing of all aspects is usually impossible – especially for reactive systems. Coverage criteria are widely adopted means to measure test suite quality. There are many kinds of coverage criteria (e.g. focussed on data flow or control flow) [11, 20]. Test generation can be stopped if a selected coverage criterion is satisfied. During our cooperation, we used different structural coverage criteria on UML state machines as a test generation stopping criterion.

There are several publications that present experience reports of model-based testing. For instance, Pretschner et al. [21] present an experience report on model-based testing. They include many aspects of model-based testing such as comparing coverage with error detection and model coverage with code coverage. In [11], Utting and Legeard present several reports on model-based testing. Their focus, however, is on the test generation technique and not on the acceptance thresholds when introducing model-based testing as a new testing technique.

There are many commercial model-based test generators for UML state machines available. For instance, the Smartesting Test Designer [3] supports the satisfaction of All-Transitions. Rhapsody ATG [22] is capable of creating test suites to satisfy MC/DC. Further commercial test generators are listed and compared in [23]. In this report, we applied the free test generation tool ParTeG [5] and the commercial test generator TestDesigner [3].

9 Summary

In this paper, we reported on our efforts to introduce model-based testing as a new testing technique in an industrial context. The results of the presented pilot project are that the introduction of MBT causes costs in the beginning. After establishing the necessary basis, however, MBT provides many advantages like automatic test design or reduced maintenance costs by fast response to requirements changes. Finally, we experienced that the customer’s main point of interest for applying MBT is not the set of features (e.g., supported coverage criteria) provided by the test generator, but integrating MBT in the test process at all. Thus, it seems like the industry may already be aware of the possible benefits of MBT but fears the issues and costs of its integration.

References