An Evaluation of Model-Based Testing in Embedded Applications

Stephan Weißleder, Holger Schlingloff
Fraunhofer-Institute of Open Communication Systems FOKUS
Berlin, Germany
{*stephan.weissleder*|*holger.schlingloff*}*@fokus.fraunhofer.de*

***Abstract*—Testing is one of the most important quality assurance techniques for software. Automating the test design allows for automatically creating test suites from high-level system descriptions or test descriptions. Furthermore, it enables testers to automatically adapt the test suites to potentially recently changed descriptions. In model-based testing, models are used to automatically create test cases. Case studies report of an effort reduction in test design between 20 and 85 percent.**

In this paper, we report on a pilot project for introducing model-based testing in an industrial context. For such a pilot project, it is necessary to adapt the existing workflows and tool chains, to train the staff, and to adapt the assets of the company. The goal is to show the full applicability of the technique at the customer site. We present the evaluations, the lessons learned, and compare the efforts of model-based and manual test design. This paper is not a 'scientific' paper in the sense that the results are reproducible for other projects and domains. Instead, our intention is to provide guidance for setting up similar evaluations.

***Keywords*-model-based testing, tool integration, industrial pilot project, experience report, case study**

I. Introduction

Quality is one of the most important aspects of today's systems. It is a marketing argument, it helps in establishing long lasting relationships to customers, and it is important for getting products certified. Quality assurance is a necessary means to support and achieve these targets. Testing is one of the most wide-spread forms of quality assurance [1], [2]. In many companies, however, testing has been and still is a tedious, manual, and error-prone process. Due to changes in requirements, changes in staff, or found defects, and the resulting additional work, this manual process causes high costs. Automation is the key concept to reduce these costs. It helps in automating boring and ever-repeating activities, in increasing efficiency and effectivity, in bringing objective results, and in motivating people. There are many concepts of test automation. For instance, test execution should be automatic. The corresponding advantages are obvious. If it comes to test design, there are several concepts such as data-driven testing or keyword-driven testing that allow for the re-use of abstract test scripts for different concrete data or instructions [3], [4]. Model-based testing (MBT) is the state of the art in automating test design [5], [6]. In MBT, models are used to (automatically) design test cases. There are initial costs for setting up the MBT framework including a modeling tool, a test design tool, and their integration in the existing tool chain. Furthermore, the staff has to be trained on the new technique, the workflows have to be adapted, and the focus on the company's assets has to be changed, e.g., from test cases to models. There are, however, several case studies that show that the application of MBT pays off despite these initial costs [7], [8].

Such success reports are often not enough to convince a test manager to apply MBT in an ongoing project. For instance, the software testing survey of Haberl et al. in Germany, Switzerland, and Austria [9] shows that only about 5% of the participants apply MBT. Potential reasons are often that ongoing projects are already behind schedule and that there is no time or budget to invest in new technologies. Furthermore, positive results from other domains are no guarantee for a positive result in the own company — even reports from other companies in the same domain or other departments in the same company are no exception. As a result, the only way to convince the responsible managers is to show that the methods will work in their own projects. A possible process of doing this includes running a market study about existing MBT tools, roughly evaluating all tools, closer investigating the top three of them, and running a pilot project on a small system of the company that is not on the critical path [4].

This is the setting that our paper reports about. Due to non-disclosure reasons, our industrial partner chose not to reveal its identity. In this paper, we report on the initial situation at the company, on the process of preparing and running the pilot project, and on the results and the lessons learned. We present both positive and negative results: things that went well and also things that did not. Before presenting everything in detail, we sketch the lessons learned: First, we measured our efforts in order to be able to estimate the costs of introduction and the productivity gain. We collected these numbers and compared them to experience data about manual test design effort from previous projects. The result is that in the worst case (if there are extensive migration activities) the introduction of MBT pays off after five iterations of test generation only. In the best case (if MBT is used right from the start) there is an immediate benefit since the time for modelling is shorter than that for hand-coding test cases. Second, automatically generated tests are not necessarily executed automatically. Although this option is known in theory [10], our experiences are that

	M0	WP1	M1	WP2	M2	WP3	M3	WP4	M4	WP5	M5
a		Analysing the Models									
b				Tool Evaluation							
c						Protocols					
d						Test generation					
e						test intents format					
f						Model transformation					
g								Training			
h										Presentation & Report	

Figure 1. Project plan for the model-based testing pilot project.

many people still assume that model-based test generation always results in automatically executable test cases [11]. In our scenario, the generated test cases are meant to be read and executed by human testers. This results in new test management challenges; for example, if issues in the model are identified after the test cases have been manually executed. Third, we describe the process aspects that were important for making this project a success. We give information about the tool support and its limitations. We also go into the technical solutions that we realized to overcome these shortcomings. Semantic-preserving model transformations as proposed in [12] to simulate coverage criteria like proposed in [13] were successfully applicated in this project.

This paper is structured as follows. In Section II, we describe the initial situation and the motivation to start the pilot project. We list the most important evaluation criteria for model-based test generation in this particular environment in Section III. In Section IV, we present the tool evaluation for preparing the pilot project. Afterwards, we report in Section V on the pilot project including the created MBT tool chain and our technical glue to make it work. In Section VI, we show some numeric results about the generated test suites and compare the costs of manual and model-based test design. The lessons learned are presented in Section VII. Section VIII contains the conclusion, a discussion of our approach, and planned future work.

II. Initial Situation

In this section, we describe the initial situation of our project. The intention was to investigate the applicability and the advantages of automatic model-based test generation for quality assurance of embedded applications. Model-based testing has been claimed to be able to automate the whole test generation process and to reduce the costs for test design by up to 85 percent [7]. The motivation for this project was to evaluate these promises and the power of model-based testing in a particular industrial context, dealing with electrical machines containing embedded control components.

Initially, we defined a project plan that describes the basic steps of the project, see Figure 1. In the first work package (WP1), the existing models were analyzed. For the sake of heterogeneity, two different models (UML state machines) were selected for the pilot project: a small one and a complex one. In the second work package, we evaluated model-based test generators and selected one of them for the pilot project. For this, we created a decision matrix that contains evaluation criteria for a tool and its surrounding tool chain. Using this decision matrix, in WP2 we evaluated several model-based testing tool chains, selected the top three of them, and jointly selected one of them. WP3 contained the main part of the project. As Figure 1 shows, we applied test generation together with model transformations that were necessary to compensate for missing features of the selected test generator. The results of the test generation process had to be delivered in a format defined by the established testing processes in the company. In order to compare MBT to manual test design, we measured all efforts during this phase. To introduce MBT to a wider circle of people at the company, we also conducted a hands-on-training in WP4 and presented the results of the whole project in WP5.

A. The Customer's Expectations

From the beginning there were detailed expectations about the interfaces of the test generator, i.e., the models and coverage criteria to be used, and the expected format of the generated test cases. The intended input of the MBT tool chain consisted of drawings in Microsoft Visio that were similar to UML state machines. The MBT tool chain should be able to import such Visio diagrams. For test generation, it was most important to be able to cover all transitions. Each outgoing transition of a composite state, however, should be fired from each substate of the composite state. Since this variation of the All-Transitions coverage criterion [5] is not supported by common test generators, we proposed to apply model transformations to flatten the state machine: Covering each transition of the flattened state machine has the same effect as the required coverage criterion [13].

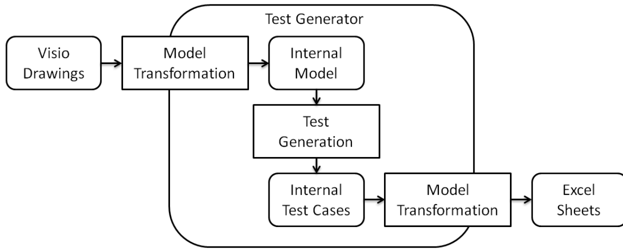

Figure 2. The intended MBT tool chain interfaces.

Finally, the generated test suites should be Microsoft Excel documents that can be automatically imported into IBM's Rational DOORS for further processing by human testers. Figure 2 shows interfaces for the input and the output of the test generation. They are used to integrate the MBT tool chain into the existing workflows. Additional model transformations are possible, e.g., to transform the Visio diagrams into the models required by the test generator.

Besides integrating the MBT tool chain into existing workflows, measuring the effort for creating the tool chain, for creating and adapting the models, and for running the model-based test generation was important. To this end, test generation for just one model does not give significant results. For this reason, we ran the test generation in different scenarios. Firstly, the impact of the model size onto the efforts was to be measured. Thus, test generation was to be performed for a comparatively small and a comparatively complex model. Secondly, the efforts for the initial test generation and for maintenance was to be measured. For that, a list of changes which are typical during a development was made; it was to be used as a pattern to modify the models in the project. Thirdly, the test generation was to be run on the original models and on transformed models. As shown in the subsequent sections, the transformation comprised the flattening of models and the semantic-preserving resolution of history states. Especially the first transformation was used to satisfy the required coverage criterion. All in all, the combination of these dimensions results in eight different scenarios that were to be investigated.

B. *Used Terms*

In this section, we introduce three terms that are used in this context. A *test intent* is an abstract test case description at model level. *Test intent steps* denote single transitions with preconditions and postconditions. *Test intent sequences* describe a sequence of test intent steps starting at the initial pseudostate of the state machine. The engineers were particularly interested in the export of test intent steps and test intent sequences based on automatically generated paths in the state machine. Correspondingly, these terms are used for the evaluation in Section VI.

III. Tool Evaluation Criteria

In this section, we present the tool evaluation criteria that were used to select a test generator.

In order to estimate the efforts of integrating MBT into the existing design processes, we evaluated complete tool chains rather than single test generator tools. All in all, we identified 42 relevant tool selection criteria by interviewing the customer. For that, we organized three half-day meetings. The tool selection criteria were grouped according to common topics like, e.g., supported features of a modeling language. Each criterion was ranked with respect to its importance on a scale from zero to ten, with an additional flag that declares whether the current criterion is indispensible, i.e., whether a violation would be a show stopper (marked with a **(*)** below). In the following, we present grouped lists of all evaluation criteria.

A. *Supported Features of the Modeling Language*

The criteria that are aimed at the supported features of the modeling language are focused on supported elements of the Unified Modeling Language (UML 2.x) [14]. The following evaluation criteria of this group were of utmost importance:

- **Support of UML State Machines*** is satisfied if the investigated test tool supports test generation based on UML state machines. Since state machines are widely used within the company, this criterion is one of the most fundamental ones.
- **Support of State Hierarchy** is fulfilled if UML state machines with a hierarchy, e.g., expressed by composite states, can be used for test generation. State hierarchy is important for concisely describing complex content. If the test generator does not support this criterion, model transformations can be used as additional preprocessing steps [12].
- **Support of History States** is satisfied if history states in the model can be processed by the test generator. History states are an important means, e.g., to change between different modes. They can also be implemented by applying model transformations.
- **Support of Inter-Hierarchy Transitions** is given if a transition can start at one state hierarchy level and end at a different one, e.g., when entering a composite state. Such transitions can be implemented, e.g., using entry and exit points at composite states. However, they can also be supported by flattening the state machines, i.e., reducing the whole state machine to one hierarchy level.

Another evaluation criterion is the support of time (importance level 5), e.g., with the UML class *TimeEvent*. Rather unimportant criteria (importance level 1-2) are in this context the support of parallel systems and their synchronization, the support of UML activity diagrams, the automatic flattening of models, and the import of models from 3rd-party tools.

B. Supported Quality Criteria at the Model Level

Automatic test generation often faces the problem of a possibly infinite test suite that could be generated. To limit the size of the generated test suite and also to steer test generation, test quality criteria, i.e., test generation stopping criteria have to be supported by the used test generator.

- **High-Level Transitions (transitions outgoing from composite states) Have to be Used From Every Substate of the Composite Source State.*** This criterion demands that every outgoing transition of a composite state is fired from every substate of this composite state. Since it can also be reached by flattening the state machine and traversing all transitions, model transformations can be applied where the test generator fails.
- **Wide Range of Supported Quality Criteria.*** This evaluation criterion simply demands that the chosen test generator should support a maximum number of quality criteria, i.e., coverage criteria. Obviously, this is no absolute evaluation criterion and can only be used to compare two test generators.
- **Transition Coverage*** should be the minimum coverage criterion the test generator satisfies. This corresponds to the popular belief that covering all transitions is the minimal acceptable coverage criterion.
- **Support of Manually Defined Sequences*** is a valuable means to integrate already existing test cases into automatic test generation. It can also be used to reduce test effort because no new test cases are about to be generated for parts of the model that are already covered by the manually defined sequences.
- **Multiple Condition Coverage** is the strongest of the condition-based coverage criteria. The support of this criterion was considered to be desirable. Using model transformations like presented in [13], this coverage criterion can also be satisfied by covering all transitions of the transformed model.

Furthermore, the following minor evaluation criteria were listed: The support of decision coverage, path coverage [4], modified condition / decision coverage [15]. All of these criteria got an importance ranking of five. Sneak path analysis [16], traceability of model elements to requirements, and the integrated measurement of code coverage received an even lower importance level. It may come as a surprise at this level that requirements traceability was considered to be of relatively low importance. This was caused by the fact that the models used are considered to be requirements by themselves. So, there are no fine-grained requirements to which model elements can be traced, and requirements traceability is not vital in this context.

C. Support of Extensions

In this part, we present the importance of tool extensions.

- **Understandability of Test Cases at Model Level** is important to easily understand and recognize the test cases at model level and the paths that are covered by them. In its most mature form, this kind of traceability should be supported for both directions: from test cases to the covered transitions and from transitions to the covering test cases.

Furthermore, we identified two additional criteria: support of debugging at model level and support of explicit variant management [17], e.g., by connecting the used models to feature models. Both of them, however, got a minimal importance mark.

D. Support of Exchange Formats / Tool Interaction

Test generators have to be integrated into existing tool chains. The minimal requirements for this are to support the import of the used models and the export of the generated test cases to the given format. Here, we present the identified evaluation criteria for tool interaction with the test generator via supported exchange formats.

- **Importing Models From Microsoft Visio*** is an important feature for the intended tool chain. Since the current models are all created in Visio, it is of major importance to support the model import from this tool.
- **The Output Format of the Generated Test Cases*** is a format for Excel sheets defined by the people at the company. As a consequence, adaptability of the output format must be supported by the test generation tool.
- **The Import of Test Cases Into DOORS*** is important because the generated test cases should be used as requirements for manual test execution. Since DOORS supports the import of Excel sheets, satisfying the previous criterion implies satisfying this one.

Additionally, we identified the need for a more abstract test modeling language, i.e., the wish for more than one supported modeling language. This is motivated by the fact that the modeling languages supported by existing test generators are often close to programming languages, but some of the intended users are not very familiar with programming. This evaluation criterion is given an importance grade of eight.

Furthermore, we identified several tools for which tool integration is important. As stated above, the tool interactions with DOORS for requirements management and Visio for modeling are considered most important (show stopper). The interactions with other tools for test management, test execution, debugging, and bug tracking are also mentioned, but considered unimportant in this project.

E. Tool Provider

In the previous parts, we described desirable features of test generation tools. The characteristics of the tool provider, however, are also of interest. Readers are probably familiar with the fate of ATG [18], which was bought by IBM without any recognizable development activity since 2009.

Customers of such tools invested a lot of money into the tool integration and now face the problem of comparatively low support and no development. Typical issues of tool integration are also described in [4]. Here, we present the evaluation criteria of the tool providers. There was only one show stopper criterion: the applicability of the tool for commercial products. Since we also evaluated open source tools, one has to be aware of open source licenses that prohibit the commercial use of the software. Other evaluation criteria are the support in form of coaching and trainings, available hotline support, implementation of customer-specific needs, the maturity/stability of the company, the license and maintenance fees, and the installability and maintainability, which is again especially important for open source tools.

IV. Tool Evaluation

After identifying the evaluation criteria for model-based testing tools, we selected several tools and categorized them. In [19], a comparative study of nine MBT tools has been performed. However, since the tools themselves have changed in the meantime and the evaluation criteria in our project were quite different, these results could not directly be transferred. In this section, we present the groups of evaluated tools. Then, we name the commercial and academic test tools which were most interesting in this context. Finally, we show the results of our tool evaluation.

A. Model-Based Test Data Generators

Model-based test data generators are focused on generating test data from models. They use an abstract descriptions of the interfaces of the system under test to generate input data for tests. As a result, the focus is not on behavioral descriptions, but on input and output data. One tool of this category is the Classification Tree Editor (CTE) [20]. It is based on identifying classifications and classes of a system and helps in selecting test cases that cover many different classes. The CTE is also able to generate such test cases automatically. Since behavioral test generation is not in the focus of model-based test data generators, and, in particular, state machines are not supported, such approaches are not considered any further here.

B. Model-Based Test Case Editors

Model-based test case editors allow for editing test cases on a model level. One example tool of this category is Time Partition Testing (TPT) from PikeTec [21]. TPT supports the modeling of test cases that are supposed to be executed on embedded systems. Furthermore, test execution and test evaluation are supported. Since these tools focus on the editing of single (abstract) test sequences rather than the generation of test suites, they are also not considered here.

C. Model-Based Test Generators

Model-based test generators generate complete and executable test suites from behavioral models. We considered several commercial and academic model-based test generators. We did not aim at a comprehensive survey (as opposed to [19]), but wanted to find the "optimal" tool for the intended purpose. Thus, our choice was guided by available documentation and personal knowledge. For instance, we had a closer look at the following commercial tools: Microsoft SpecExplorer [22], the TestBench of imbus [23], MaTeLo of All4Tec [24], the Conformiq Designer [25], the ATG of IBM [18], the Smartesting CertifyIt [26], and the MBTSuite [27] of sepp.med and AFRA. We installed and ran as many of these tools as possible. For the Smartesting tool, it was not possible to get an evaluation license so that we had to stick to the existing product descriptions. We also investigated the following academic test generation tools: the Partition Test Generator (ParTeG) [28], Azmun [29], MISTA [30], and modelJUnit [31].

D. Results of the Tool Evaluation

After a first rough evaluation of the tools, some of them were excluded because certain evaluation criteria were not met. As the result of this first evaluation, Conformiq Designer, sepp.med/AFRA MBTSuite, and Smartesting CertifyIt were considered to be the currently most interesting commercial test generation tools. To us, the top three academic test generation tools currently are ParTeG, Azmun, and modelJUnit. Please note that this is by no means a general comparison or absolute rank list of model-based testing tools. Instead, we identified these tools to the best of our knowledge as the most fitting tools for the setting defined in this special project. The limitations of the tools that are not listed here were often caused by their focus on a different test generation approach. Mainly, this concerned the use of other modeling languages than UML state machines. The six tools mentioned above were investigated quite closely — all evaluation criteria were assessed for each tool and corresponding tool chain.

One first result was that the import of Visio diagrams was not supported by any of these tools (state: March 2012). Reasons for that are, e.g., violations of data flow information, violation of UML syntax, or the use of model elements to represent a legend in the Visio drawings. Consequently, the automatic import of Visio diagrams could not be used as a distinguishing criterion for our evaluation. Therefore, we decided to create the models manually from Visio drafts using Enterprise Architect 8.0 from Sparx Systems.

We evaluated all tools in two different settings: In the first setting, the capabilities of the test generators were evaluated in isolation. In the second setting, a model transformator was added as a pre- and post-processor to the evaluated tools. Thus, it could be evaluated how this tool chain integrated into the available tool environment.

The degree of compliance to an evaluation criterion (from 0=noncompliant to 10=fully compliant) was multiplied with the importance level of the criterion to give a ranking for this criterion; the final score for each tool was obtained by adding up all criteria.

When comparing the scores for commercial and academic tools, commercial tools were almost always better than academic tools. This was not necessarily caused by better or more mature test generation approaches. Instead, the commercial tools scored in criteria like integration with other tools, tool support, and exchange formats. In theoretical aspects like, e.g., supported coverage criteria, academic tools often received better scores. The advantages of the commercial tools are, however, decisive for practical application.

After considering the resulting scores, it was decided to use Conformiq Designer for the pilot project. Again, we emphasize that this is not a universally valid ranking of MBT tools: Compared to the other top three tools, the selection of this tool was mainly caused by its support of state hierarchies, time, and a variety of coverage criteria - even if not all elements of UML state machines are supported.

V. Pilot Project

Based on the tool selection in the previous section, we started the pilot project. In order to compare the efforts for "small" and "big" tasks, we used two different state machines for test generation, a small and a complex one. Furthermore, in order to deal with model transformation preprocessing, we additionally considered "flattened" versions of these machines, which contain no hierarchical or parallel submachines. Finally, we ran the test generation also for a "version in maintenance" of these machines, i.e., where we made some changes which are typical for those occurring frequently in the design process. In this section, we describe the course of actions during the pilot project.

1) Adapting the Models: As described above, the Visio diagrams given as examples for the study could not be used directly as input for the test generator. Thus, we first had to translate them manually into UML state machines. Besides translating the given sample diagrams into UML, for our analysis we also had to add a few Conformiq-specifics. For the interested reader, we give a short overview of the necessary steps: **a)** Classes of stereotype *record* have to be created for all sent or received messages. **b)** There has to be one class that contains a state machine representing the behavior of the system under test. One composite structure diagram has to be created in which all ports to this class must be defined. **c)** To connect a received message with a trigger of a transition, the trigger has to have the following format: *<port>:<class>*. For instance, if a message of type *M1* is received via port *in*, then the corresponding trigger has the name *in:M1*; Attributes like, e.g. *x*, of the message class are referenced via *msg.* in the guard like, e.g., *msg.x = 0*. **d)** To send messages, an object of the message class has to be created. This class is then sent via the corresponding port with the following construct: *<port>.send(<msg-object>);*. For instance, to send a message of type *M2* via the port *out*, the effect of the transition has to have the following content: *M2 o; out.send(o);*. **e)** For the developed scripting backend, states have to be explicitly marked that are important for the export. This can be done by using *scenario* informations. If, for instance, a visited state has an entry action with the text *scenario "door closed";*, then the output of the scripting backend is going to write *door closed* when this state is part of a test sequence generated by the test generator. **f)** The model has to contain a class *System* that contains one main function that calls the class that contains the state machine.

2) Importing the Models and Test Generation: These models we designed in Enterprise Architect could be imported in Conformiq Designer using the Enterprise Architect dialect of XMI 2.1 as an interchange format. Afterwards, we could generate test sequences with the Conformiq Designer. For this, we selected the coverage criteria All-States, All-Transitions, and Conditional Branches [25].

3) Exporting Test Cases: We created a scripting backend that translates the generated test sequences directly into Microsoft Excel files that contain the test intents in the format required for further processing. This scripting backend was developed in Java and used an interface predefined by Conformiq.

4) Creating Model Transformations: We implemented a model transformator that flattens state machines (reduction of hierarchical state machines to one level) and resolves the history states while preserving the behavior of the state machine. We used a conversion script of openArchitectureWare to transform the Enterprise Architect file into an ECore-UML-2.1 file. This file was read using the Java-based ECore implementation. We implemented the model transformation in Java using the ECore: The flattening of the models is implemented by copying each outgoing transition of a complex state and pasting it to each contained state. To resolve a history state, a new variable has been defined that "remembers" the formerly visited substate and controls the selection of the substate when the complex state is re-entered. Finally, the ECore models were transformed into the four typical files used and read by Conformiq Designer.

5) Measuring the Effort: We measured the effort for all of the described activities. Numbers are rounded to full hours, except for model maintenance, where the efforts consisted only of few minutes. The figures are valid for an experienced engineer and do not include a "learning curve".

VI. Cost Comparison of Manual and Model-Based Test Design

In this section, we present and analyze numerical data on the generated test suites, our efforts, and a comparison to the effort of manual test creation.

Model	Transformed?	Changes applied?	Tests in Conformiq	Test steps in Conformiq	Test Intents	Test Intent Sequences	Test Intent Steps
Small			4	16	12	4	8
Small	X		8	32	20	8	12
Small		X	6	27	18	6	12
Small	X	X	8	35	22	8	14
Complex			36	275	87	36	51
Complex	X		101	815	212	101	111
Complex		X	36	281	87	36	51
Complex	X	X	101	834	213	101	112

Table I
THE SIZES OF THE GENERATED TEST SUITES.

Action Id	Task	Effort (hh:mm)
1	Model creation small in Enterprise Architect	01:00
2	Model creation complex in Enterprise Architect	08:00
3	Model maintenance small in Enterprise Architect	00:05*
4	Model maintenance complex in Enterprise Architect	00:05*
5	General introduction to tool interactions	08:00
6	Implementation of the scripting backend for the Excel output	12:00
7	Implementation of the model transformator	45:00
	Manual design of a test suite small	01:00*
	Manual design of a test suite complex	16:00*
	Manual maintenance of a test suite small	01:00*
	Manual maintenance of a test suite complex	04:00*

*: approx.

Table II
EFFORT FOR THE SEPARATE TASKS (APPROX.)

As described in Section II-A, we generated test suites for two models, a small one and a complex one in eight different settings. The scenarios and the number of generated test artifacts are presented in Table I. This table shows for each scenario the number of test cases and test steps (interactions with the environment) as automatically generated by Conformiq and the number of test intents, test intent sequences, and test intent steps as created by the backend script that was integrated into Conformiq for the specific output (see terms in Section II-B). It also shows that the test suites are bigger for transformed (flattened) models.

A goal of this study was to measure the effort for all tasks during implementing and running MBT. In this section, we present the measured numbers and compare them to numbers for manual test creation. For this, we refer to the Test Intent Sequences of the transformed models in Table I. In the first step, we measured the effort of creating the models, changing the models, and implementing the MBT tool chain. Table II depicts the efforts for the executed steps in hours and minutes.

These numbers show that creating a model can take several hours. However, for an experienced engineer this is still faster than the manual design of an equivalent test suite. For the complex model, manual creation of the test suite took two days.

The maintenance, e.g., by making small changes to the model, results in an effort of only a few minutes. (One has to mention that the maintenance effort comprises here only the actual work of changing the model; thinking about which parts of the model have to be changed probably takes longer.) In contrast, updating a manually created test suite is a non-negligible task; the estimation is given below.

We assume that changing one model on average takes about 5 minutes. As we will see, the concrete number (1 to 10 minutes) is unimportant for the final result - the dimension matters. The initial effort for creating and setting up the tool chain takes about 65 man hours (sum of actions 5, 6, and 7). Due to the reusability of the tool chain, it may be a bit “unfair” in a comparison to assign all MBT tool chain costs to one project only: For single applications of MBT, users would use the given front-end and only have to implement the scripting backend (action 6). For more frequent applications of MBT, it is necessary to integrate the MBT tool in the existing tool chain. This step required once a significant effort (actions 5 and 7) for the sum of all projects. Since this effort is only necessary, however, if MBT is to be applied to a larger set of projects, the average effort for a single project is significantly reduced. In our case, we estimated one hour for the tool chain effort, which lead us to an estimated effort of 13 hours for this project.

In order to compare manual and automated test design efforts in maintenance, we have to estimate the average effort

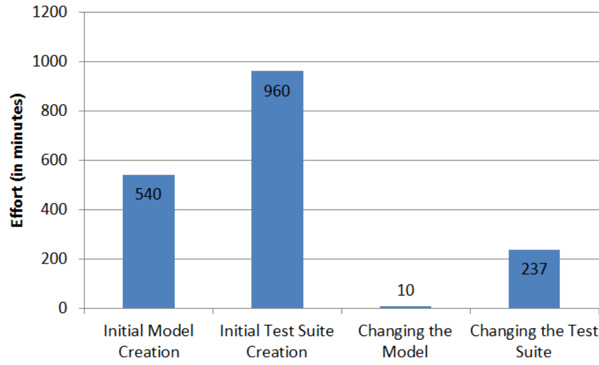

Figure 3. The single effort for one iteration.

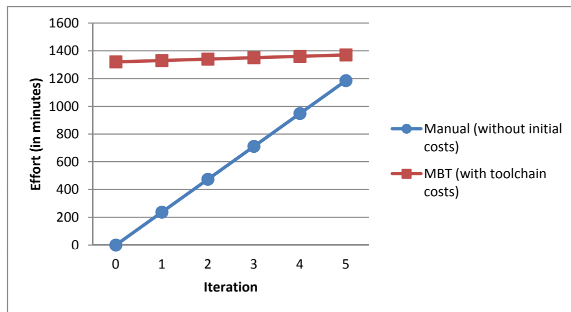

Figure 4. MBT is introduced in a running project with tool chain costs.

of manually changing test suites. The generated test suite for the complex flattened model consists of 101 test cases. It took two days for manually creating the test suite for the complex model. This results in a manual test design effort of approximately nine minutes per test case. Thus, if the test case needs to be adapted, we calculate an effort of nine minutes. We estimated one minute per test case for checking whether this test case needs to be changed or not. Furthermore, there are documents that showed the average changes during one iteration. In most iterations, there were only around four small changes that consist of, e.g., adding a transition or changing a guard. We ran a few of such

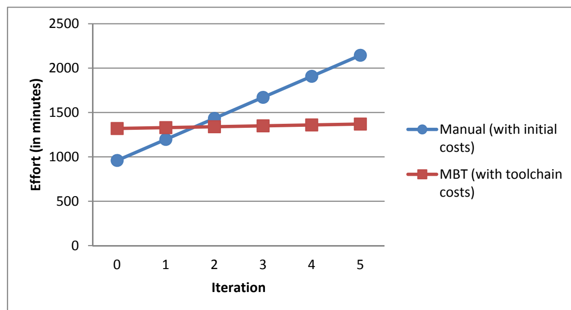

Figure 5. MBT with tool chain costs and manual tests with initial costs.

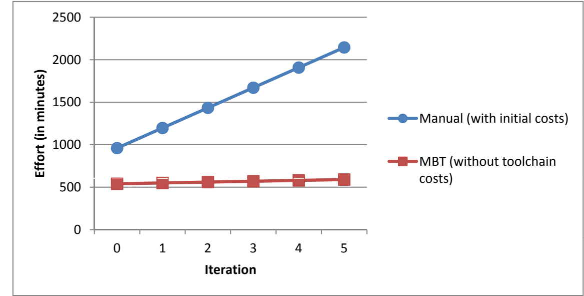

Figure 6. A new project without a test suite and with no tool chain costs.

alternations on the model and regenerated the test suite. Afterwards, we identified the test cases that were changed. For the complex model, 11 out of 101 test cases had to be replaced. For the small model, five out of eight test cases had to be replaced. All in all, this results in 16 test cases that had to be replaced and 93 test cases that could be left unchanged. Assuming the effort of nine minutes per changed test case and one minute per unchanged test case, this results in a manual effort of 237 minutes per iteration or changes in the complex model, respectively. In contrast to these estimated manual efforts, we measured a total of less than ten minutes for changing the models and automatically generating new test cases.

Based on these data, we are able to compare the effort for applying MBT and the effort for manually deriving test suites. Leaving aside the fact that manual test creation is an error-prone activity, this comparison does not take any quality criteria into respect (except the proviso that both the automatic and the manually created test suite had to satisfy the required model-based coverage criteria). We assume, however, that the application of stronger coverage criteria than All-Transitions results in bigger test suites and, thus, in an increased test effort with an increased savings potential by applying MBT.

In the following, we present figures that depict the different efforts for manual and automatic test creation. Figure 3 shows a comparison of the efforts for the initial model creation and the initial test suite creation, and a comparison of the efforts for changing the model and for changing the manually created tests for one iteration. We considered three scenarios:

1) A manually created test suite does already exist and MBT is newly introduced in an ongoing project. This means that there are no initial costs for manual test creation but costs for MBT tool chain creation. Figure 4 shows the comparison for Scenario 1. It shows that MBT pays off after the fifth iteration.

2) There is no test suite for the considered project, yet, which also results in initial effort for manual test creation (here: two days). Figure 5 shows the comparison for Scenario 2. It shows that due to the manual test creation

effort, the application of MBT already pays off in the second iteration.

3) There is no test suite for the considered project, yet. The tool chain, however, was invented during a previous project and can be reused for the current one. Figure 6 shows the effort comparison for a new project for which we assume that the tool chain already exists and there is no test suite, yet. It shows that MBT pays off right from the start.

VII. Lessons Learned

After finishing the pilot project, we presented the results to the management. We also held an MBT workshop for requirements experts and software engineers within this context. In both the presentation and the workshop, we got additional feed-back. Here, we present the lessons learned.

Firstly, we measured the efforts of the model-based test creation and compared it to the efforts of manual test generation. Generally, the big benefit of MBT is the low cost for regenerating a test suite after changes have been made to the requirements. If the requirements would never change during the course of a project, then manual and automated test design are of comparable complexity. However, this assumption is unrealistic: in all projects we have been involved so far, there were frequent changes and additions to the requirements, so test suites had to be revised over and over again. In this case, MBT will always be more efficient than manual test design. If MBT is newly introduced in a company, and there are only a few projects which will use it (so that setting up the tool chain is an important cost factor), or there already is a significant amount of manually designed test cases, then MBT will pay off only after a few such iterations. Otherwise, if MBT is introduced in a new project or on a broad scale, then it pays off right from the beginning.

Secondly, the application focus has a strong influence on the evaluation criteria and results. There is no such thing as "the" best MBT tool for all purposes and application areas. In our scenario, the results of the automatic test generation were not intended to be used for automatic test execution. This was mainly caused by the physical system interface. Often, test design is one of the most time-consuming activities during testing. In the present setting, however, the manual execution of the test cases caused the main testing effort. In MBT, issues arising during test execution can lead to changes in the model; as a consequence, test cases have to be regenerated. It is important that test cases which were already executed should be be left unchanged by this regeneration as far as possible. Otherwise, the repeated execution of similar tests could cause unnecessary costs. Therefore, existing tests have to be taken into account during test generation. There are approaches to include manually generated test sequences into model-based test generation [32]. However, since this was not the focus of our study, we did not take these approaches into respect.

Thirdly, we observed that introducing a new technology that spans the working areas of several departments can lead to a better communication of the engineers within these departments. We organized regular meetings and in the final presentation, we got the feed-back that all departments considered MBT to be a powerful tool. The most important advantage as considered by the participants of our meetings was the early review and validation of requirements. As a necessary premise, however, all participants have to be able to understand the models and if necessary have to be trained.

However, we also identified several issues for the application of MBT. For instance, for people outside of computer science, the current tools often are too much programming-like, although applied at the model level. As already mentioned, the engineers have to be trained on the modeling language and the supporting tools. Tool integration still is an issue. Several items had to be defined multiple times in different places in the presented tool chain (for example, the names of events at transitions and the corresponding message classes with the same name [25]). Here, the definition of a more abstract domain-specific language can help to hide such test tool-specific definitions.

Finally, this project showed that the simulated satisfaction concept defined in earlier work [13] is a powerful tool in real projects. In academic tools, such concepts can be incorporated easily, whereas for commercial tools, some "workaround" has to be found. In general, academic methods and tools often provide more functionality like, e.g., the set of supported coverage criteria. Commercial tools, however, offer more support for industrial application like, e.g., the integration into existing tool environments.

VIII. Conclusion, Discussion and Future Work

In this paper, we presented our experiences of introducing model-based testing in an industrial context. We described the setting, the expectations of our customers, and the introduction of MBT. For this we concentrated on describing the tool evaluation, the pilot project, and the lessons learned. Furthermore, we showed our experiences on the efficiency of MBT: We measured the efforts for automatic and manual test creation and concluded that even taken initial costs into respect, MBT pays off after a few iterations.

Although the results of the present report are promising, they should always be seen within their context. Firstly, this paper describes one small case study only and is probably not representative for other domains or companies. However, it can be useful for setting up similar evaluations. As related case studies show, the impact of applying MBT is varying from domain to domain [8]. We think that this study is another helpful piece to substantiate the advantages of MBT. Secondly, the consideration of tools was not exhaustive and may have been biased, in particular, since the first author of this paper is also the author of a model-based test generation tool. However, in MBT the tool landscape is

quickly changing, and there are no "objective" criteria for the assessment of MBT tools. Peleska [33] created an embedded systems testing benchmarks site where different tools can be compared on the same model; this might improve the situation.

Future work includes the investigation of MBT integration in different design processes. In particular, we showed that new test management issues may result if automatically generated test cases are manually executed. More generally, dealing with changing and extending requirements is a promising research area for MBT.

References

[1] G. J. Myers, *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 2008.

[3] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] International Software Testing Qualifications Board (ISTQB), "Certified tester syllabus," http://www.german-testing-board.info/en/lehrplaene.shtm, 2012.

[5] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[6] E. J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-Based Testing for Embedded Systems*. CRC Press, 2011.

[7] I. Forrester Research, "The Total Economic Impact of Conformiq Tool Suite," http://www.conformiq.com/tei-conformiq.pdf, 2012.

[8] S. Weißleder, B. Güldali, M. Mlynarski, A.-M. Törsel, D. Faragó, F. Prester, and M. Winter, "Modellbasiertes Testen: Hype oder Realität?" *OBJEKTSpektrum*, vol. 6, pp. 59–65, 2011.

[9] P. Haberl, A. Spillner, K. Vosseberg, and M. Winter, "Wie wird in der Praxis getestet? Online-Umfrage Deutschland, Schweiz und Österreich," http://www.objektspektrum.de, 2011.

[10] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Softw. Test. Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[11] Wikipedia, "Model-Based Testing," http://en.wikipedia.org/wiki/Model-based_testing, August 2012.

[12] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability Transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[13] S. Weißleder, "Simulated Satisfaction of Coverage Criteria on UML State Machines," in *International Conference on Software Testing, Verification, and Validation (ICST)*, April 2010.

[14] Object Management Group, "Unified Modeling Language (UML), version 2.4," http://www.uml.org, 2011.

[15] J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," in *Software Engineering Journal, Issue*, vol. 9, September 1994, pp. 193–200.

[16] J. P. Rankin, G. J. Engels, and S. G. Godoy, "Software Sneak Circuit Analysis," AFNL-TR-75-254. Boeing Aerospace Co., Houston, Texas, Tech. Rep., 1976.

[17] K. Pohl and A. Metzger, "Software Product Line Testing," *Communications of the ACM*, vol. 49, no. 12, pp. 78–81, 2006.

[18] IBM (Telelogic), "Rhapsody Automated Test Generation," http://www.telelogic.com/products/rhapsody.

[19] H. Gtz, M. Nickolaus, T. Roßner, and K. Salomon, "iX-Studie: Modellbasiertes Testen," http://www.heise.de/kiosk/special/ixstudie/09/01/, 2009.

[20] E. L. (Bringmann) and J. Wegener, "Test Case Design by Means of the CTE XL," Copenhagen, 2000.

[21] PikeTec, "TPT (Time Partition Testing) Version 4.2," http://www.piketec.com/, 2012.

[22] Microsoft Research, "SpecExplorer," http://research.microsoft.com/en-us/projects/SpecExplorer/, 2010.

[23] imbus, "TestBench," http://www.imbus.de/english/produkte/imbus-testbench/, 2012.

[24] All4Tec, "MaTeLo," http://www.all4tec.net/index.php/All4tec/matelo-product.html, 2012.

[25] Conformiq, "Designer 4.4," http://www.conformiq.com/.

[26] Smartesting, "CertifyIt 5.2," http://www.smartesting.com/index.php/cms/en/product/certify-it, 2012.

[27] sepp.med, "MBTSuite," http://www.seppmed.de, 2012.

[28] S. Weißleder, "ParTeG (Partition Test Generator)," http://parteg.sourceforge.net.

[29] Siamak Haschemi, "Azmun," http://www.azmun.de/, 2012.

[30] D. Xu, "Model-Based Integration and System Test Automation (MISTA)," http://www.homepages.dsu.edu/dxu/research/MBT.html, 2012.

[31] M. Utting, "Modeljunit," http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/.

[32] D. Sokenou and S. Weißleder, "Combining Sequences and State Machines to Build Complex Test Cases," in *MoTiP'09: Workshop on Model-Based Testing in Practice*, June 2009.

[33] J. Peleska, "Embedded systems testing benchmarks site," http://www.informatik.uni-bremen.de/agbs/benchmarks/, 2013.