Automatic Traceability from Tests to Requirements by Requirement-Based Refinement

Stephan Weißleder[1], Thilo Girlich[2], and Jan Krause[3]

[1] Fraunhofer-Institut FOKUS, Kekuléstraße 7, 12489 Berlin, Germany
[2] Thales Transportation Systems GmbH, Colditzstraße 34-36, 12099 Berlin, Germany
[3] ifak Magdeburg e.V., Werner-Heisenberg-Straße 1, 39106 Magdeburg, Germany

Abstract. Requirements coverage is an important aspect in testing safety-critical systems. To measure and achieve it, test cases have to be traced to requirements. Besides measuring the degree of requirements coverage and motivating the existence of each test case, traceability also allows for prioritizing test cases based on the importance of the corresponding requirements. In most cases, however, establishing this traceability is a manual and erroneous piece of work that is going to be partially repeated for every change. For advanced test generation techniques like model-based testing, in which many test cases can be generated automatically, there is a big need to automatically derive requirements traceability.
In this paper, we present an approach and a corresponding prototype implementation for automatically tracing test cases to requirements. As major advantages, verification and validation of test suite and test model are improved, system assessment benefits from high-resolution traces to requirements, and our approach is independent from the applied test design technique. We present an example from the European Train Control System ETCS to show the applicability of our approach in industrial projects.

1 Introduction

Requirements define the functionality and capabilities of a system. For complex, large, or safety-critical systems, requirements are the legal basis for system engineering. For this reason, it is of major importance for quality assurance to show that the requirements are satisfied by the developed system. To achieve this proof of requirements satisfaction, it is necessary to create the traceability from test cases to requirements, i.e., to show that the created test cases verify the fulfillment of requirements. There are many different black-box test design techniques and many ways to preserve traceability during the test design process. Our approach, however, is focused on establishing traceability from tests to requirements independent of the test design process, which makes our approach applicable to a majority of test design approaches. Besides the most important aspects of creating and maintaining the traceability for validation purposes, traceability can also be used to improve test efficiency, e.g., by transferring the priority of requirements to test cases. This can be used, e.g., to reduce the set of executed test cases to a minimum set of important test cases.

Testing is always an important, but tedious and erroneous piece of work with different kinds of test levels and test approaches [2]. For complex systems, usually large test suites have to be created and maintained. Automation is the key to increase the test efficiency and to reduce the test effort. For automatic test design, model-based testing (MBT) [14] is the state of the art. In model-based testing, test suites are derived based on models. In the best case, this derivation is an automatic process. For safety-critical systems, the generated test cases have to be traceable to the requirements. Since automatically generated test suites can become large, it is advisable to also create the requirements traceability in an automatic manner. As we will show in Section 2, there are several approaches to create traceability for test cases that were generated by applying model-based testing. Most approaches, however, are focused on annotation model elements - a way that is in many cases infeasible and for bigger projects at least hard to manage.

In this paper, we present an approach to refine and compose requirements and to automatically link these requirements to test cases. We also refer to our prototypical implementation and substantiate our descriptions with an example from the European Train Control System (ETCS): We show models and requirements of the ETCS Radio Block Center (RBC) for a train that leaves one RBC-controlled sector and enters another one

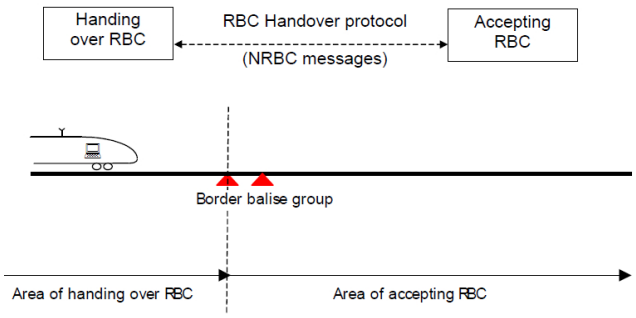

Fig. 1. The ETCS RBC/RBC handover (from UNISIG-Subset-039 [13]).

in an ETCS RBC/RBC handover (HOV) as described in UNISIG-Subset-039 [13] and depicted in Figure 1. This paper is an extract of the diploma thesis of Thilo Girlich [5]. More details can be found there.

The paper is structured as follows. Section 2 contains the related work. In Section 3, we describe refined types of requirements that are necessary to express the complex relations of requirements and tests. We show more details about the refinement of test cases in Section 4. In Sections 5 and 6, we describe our prototypical implementation and the case study in which we apply our prototype together with a test generator based on Petri net techniques [7]. Finally, we present conclusion, discussion, and future work in Section 7.

2 Related Work

Requirements engineering [11] is an integral part of system engineering. There are different types of requirements like, e.g., requirements from customers and provider requirements that meet the customers requirements. Typically, the high-level descriptions are refined, clarified, and specified in a step-wise manner with the goal to add all necessary information for the system engineer. Requirements are usually handled as objects with relations such as refine and verify, etc., and one composite relation to structure requirements [9]. To our best knowledge, however, there seems to be no approach to describe relations between behavioral functional requirements. In this paper, we define such relations. They allow for automatically tracing test cases to requirements. Our case study from the UNISIG [13] also shows the need for our defined relations.

Software testing is an essential part of software engineering [2,1]. There are many ways to run or design test cases. Model-based testing [14] is a test design technique that uses models to derive test cases. In its most mature form, test case derivation is done automatically. There are many ways to steer automatic test generation from models. Applying structural coverage criteria like All-Transitions [14] for graph-based models is a common way [15]. There are several approaches to integrate requirements traceability into automatic test generation. For instance, there are tool chains like the TestBench [6] being focused on test management that start at requirements and derive sequence and activity diagrams to represent each requirement. Such approaches, however, do not include test generation from complex behavior models like state machines, which is a powerful and wide-spread means to automatically generate test cases. Other commercial test generation tools like Conformiq Designer [3] or MBTSuite [10] use state machines for test generation and annotate model artifacts such as states, activities, or transitions with identifiers of given requirements. These annotations are often restricted on single elements of the model and do not cover complex behavior nor sets of behaviors. Friske and Schlingloff discovered a discrepancy between generated test artifacts and the expectation of testers with respect to requirements that demand for functional behaviors like repeatings and overlaps by manual analyses [4]. For instance, some paragraphs or statements in the specification UNISIG-Subset-039 [13] even require non-deterministic sets of behaviors, e.g., repeating actions without defined ends or without finite sets of actions to combine. It is obvious that these complex requirement cannot be linked by elementary artifacts such as states or transitions. As one step towards separating requirements and test generation with state machines, Weißleder and Sokenou [12] combine sequence diagrams and state machines for automatic test

generation. They decoupled the state machine and the representation of requirements. In contrast to these approaches, we do not focus on preserving traceability during the automatic test generation, but establish traceability independent of the test generation. As mentioned before, this has several advantages like, e.g., separating the test design process and the traceability management, which means that our approach can be applied to a majority of test design techniques.

3 Behavior-Oriented Requirements

In this section, we define new requirements types that are a behavior-oriented refinement of existing requirement types. These new types of requirements are extensions of the System Modeling Language (SysML) [9]. They will be used to describe the relations of behavioral specifications. We start by giving a short introduction into typical kinds of requirements and continue with our definitions of an extended SysML.

Specification documents, customer models of systems and their intended use contain different kinds of requirements. Many requirements describe basic and elementary aspects or actions of the intended system. In our example of the ETCS RBC/RBC handover from the UNISIG-Subset-039 [13], an example of a simple action is the following: *4.2.2.2: The handing over RBC is responsible to request route related information when necessary, i.e. for efficient handover when a train is moving towards a border, and is allowed to limit the amount of route related information to be received.* Such lower level descriptions are in general deterministic and do not describe interactions with other lower level descriptions. For instance, events like the loss of communication or the need for sending other messages are described in different atomic requirements.

In our approach, we presume that atomic requirements can be modeled by sequence diagrams. Although there may be examples in which this is not the case, this assumption is backed up by our case study. Even if not explicitly defined as sequence diagrams, many specifications describe actors and interfaces that allow for defining sequence diagrams. If all actors and interfaces are specified, then the entire capability of the system to interact with its environment can be described as sequences.

In many cases, however, such sequences of atomic requirements can be interruptible between two steps, i.e., message transfers. As a consequence, it is hard to define the possible overlappings of the corresponding actions and even a complete set of atomic requirements is not sufficient to specify the entire system behavior. For the mentioned example 4.2.2.2, this means the questions of how many and which other actions may be performed before the request for route related information is sent. Or how many actions could follow until the responsibility is passed to the other RBC? For this reason, models of how to arrange atomic requirements to the functionally intended behavior are necessary.

There are kinds of requirements that are made for describing complex behavior, i.e., complex interactions between actions of atomic requirements. Such complex behavior can be, e.g., unordered compositions of atomic requirements with optionality, repeats, overlaps, or concurrency. For instance, one such complex requirement from the ETCS RBC/RBC handover is the capability of an RBC to overlap two handover transactions which is defined in chapter five of the Subset-039: *5.1.2.4: The RBC shall be able to handle RBC/RBC Handover transactions which, a) ... b) overlap (i.e. RBC/RBC HOV transactions are handled simultaneously).* As described in SysML [9], each complex requirement is in general a set of compositions of atomic requirements. However, the complexity of the possible interactions of the contained simple actions is exponential depending on the number of possible interactions between simple actions [5]. For this reason, there is the need to further describe the kind of possible interactions between these actions. In the following, we define corresponding new types of requirements as an extension of SysML requirements diagrams.

During our studies of the UNISIG-Subset-039, we identified four new kinds of requirements compositions: *ContainRequirement*, *RepeatRequirement*, *OverlapRequirement*, and *ConcurrencyRequirement*. We define these composite requirements as follows:
ContainRequirements cover general statements of functionality. In some cases, a (set of) start and end requirements are covered by this type of requirement. Some atomic requirements might be optional. In the UNISIG specification, an example requirement demanding an entire process from beginning until a train is under responsibility of an accepting (ACC) RBC is such a *ContainRequirement*. An example of such a requirement is given in Figure 3.

RepeatRequirements can capture both atomic and *ContainRequirements*. It states that the described behavior of the covered requirements must be repeatable by the system. For instance, messages and entire processes between two RBCs are such repeatable requirements of the UNISIG in chapter five.
OverlapRequirements state that a system must be able to overlap captured requirements. Overlapping means that the tested system can switch the executed action. For instance, two or more trains approaching the RBC border simultaneously demand for the overlaping RBC handover transaction.
ConcurrencyRequirements capture requirements that describe behaviors that a system has to show concurrently. Communication systems such as RBCs have to be able to communicate *simultaneously* in two directions which means that they have to handle concurrency on a message, process, or computing level.

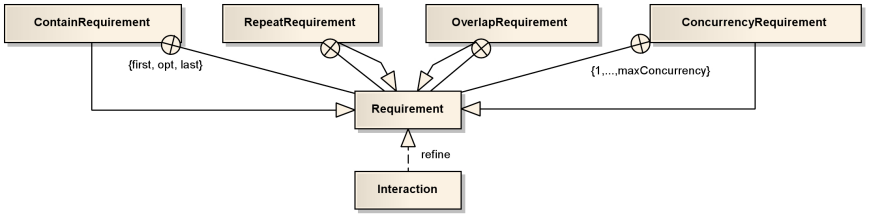

Fig. 2. Extension of the SysML requirements diagrams to enhance modeling of requirements.

Figure 2 shows the corresponding extension of the SysML with the four defined types of requirements. We use the common *arrowed* edge of UML/SysML to define that these four types of requirement are derived from the stereotype class *Requirement*. With the SysML edge that marks *contain* relationships between requirements we define that these requirements can capture sets of requirements in their stereotyped way. Figure 2 also depicts the link between *sequences*, i.e., tests defined as interactions and *requirements* in our extension of the SysML. Figure 3 shows how both mentioned requirements correspond to each other. The requirement 4.2.2.2 is an essential part of requirements derived from paragraph 5.1.2.4.

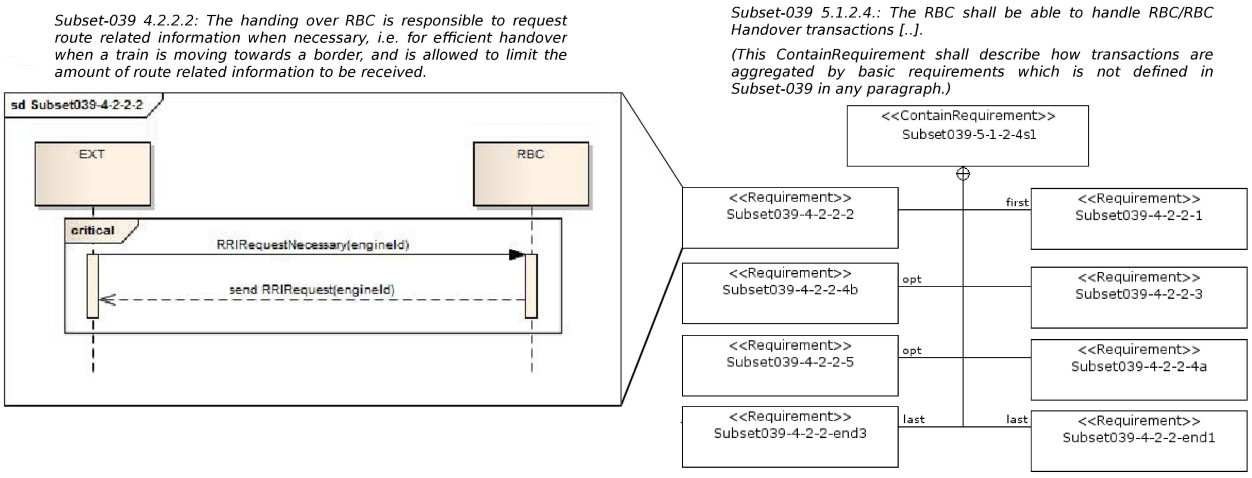

Fig. 3. Example of how the extended SysML meta model is used to define two ETCS requirements.

4 Requirement-Based Refinement of Test Cases

In the previous section, we defined several types of requirements. Here, we propose requirement-based refinement of test artifacts as a formal solution independent of test case design methods and corresponding tools. Premise of a formal refinement of test cases is requirement modeling in a combination of SysML requirement diagrams and sequence diagrams of the Unified Modeling Language (UML) [8]. Abstract test cases are usually described as sequences to illustrate a test activity and the expected behavior of the system under test. Due to the usage of the same notation, relations between requirements and test cases can be easily detected. Furthermore, requirement-based refinement, i.e., matching of tests to requirements can be implemented as a pattern matching algorithm that uses requirement descriptions as patterns.

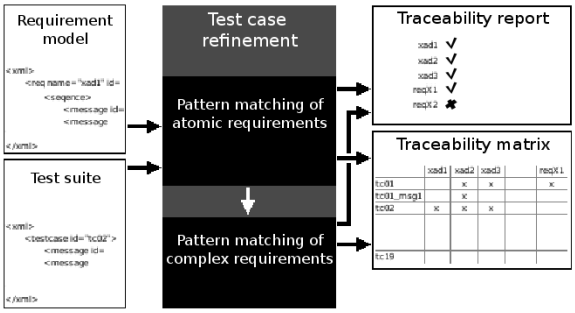

Fig. 4. Requirement-based refinement of test cases.

The proposed methodology of test case refinement is split into two major steps calculating requirement traces. Figure 4 roughly depicts these two steps as well as input and output of the refinement approach.

In the first step, atomic requirements are detected in the test case sequence via pattern matching. Behavioral descriptions of atomic requirements are detected by directly comparing the requirement-related sequence diagrams with the test cases. If all necessary messages of a requirements sequence were detected, then a trace between the requirement and the test case is created.

In the second step, complex requirements are detected by using our defined types of behavioral interaction. We assume that behaviors of complex requirements do widely vary and cannot be modeled by single sequences. As a consequence, a direct comparison of a complex requirement and a test case is no solution as it was done for atomic requirements. Instead, our method uses our defined types of requirements and the referenced atomic requirements to derive patterns for sequences that are fulfilling complex requirement. Here, the main piece of work consists of defining a new language and regular expressions that correspond to the definitions of our requirement types. These regular expressions have parameters that correspond to the referenced atomic requirements. The definition of the language and the corresponding regular expressions can be looked up in [5]. The effort for running the algorithm could be reduced by reusing the traceability links of test cases to the atomic requirements that are referenced in the complex requirement.

Although complex requirements have a high number of instances, links between test cases and specifications can be easily established using the results from the first step. In the second step, it is analyzed if the order of traces to atomic requirements found in the first step matches the rules of a complex requirement. In our solution, some kinds of complex requirements can capture other ones. This is due to *ContainRequirements* can also be required to be repeated, overlapping, or concurrent. It might also be possible that *RepeatRequirements* or *OverlapRequirements* capture other ones, but we decided to exclude this from our prototypical implementation to avoid cycles and infinite recursion.

All in all, this approach allows for linking test artifacts with different types of requirements and for creating a traceability matrix that can be used to measure test quality and to ease test evaluation after execution.

5 Prototypical Implementation

We implemented the presented methodology in a Java-based prototype. The input for this tool are requirements modeled as proposed and test cases as sequences. The output of the tool is a traceability matrix and a report of traceable / untraceable requirements and test cases. We used this prototype implementation to evaluate the methodology of refinement and the approach of pattern matching. The tool is designed to easily implement new stereotype classes to capture not yet considered kinds of requirements. In a tool chain consisting of a modeling tool and a model-based test case generator, this prototype allows to model and design tests that can be validated and managed in rapid engineering cycles. This tool chain could also deliver test cases, coverage measurement, and requirements traces. For the prototypical approach, XML was used as interface language for the requirement model and the test cases.

6 Case Study: ETCS RBC/RBC Handover

In this section, we describe our methodology and our prototypical implementation by a concrete example from the ETCS RBC/RBC handover specification. The European Train Control System (ETCS) is an approach to establish an interoperable and standardized train control system in Europe that is going to replace the national systems in the next years. For guarding a train continuously by GSM-R radio communication in ETCS Level 2, Radio Block Centers (RBC) are responsible to support trains with Movement Authorities, Emergency Stop Telegrams, and Track Information. RBCs are essential for the on-board units to calculate their braking curves based on Movement Authorities informing the on-board unit about the braking distances. Every RBC has an assigned area of responsibility, for which it controls the movement of the trains inside this area. Trains crossing the border between two RBCs have to be handed over according to UNISIG-Subset-039 [13]. Handover transactions take place between a handing-over RBC that has currently responsibility over the train and the accepting RBC in the direction of travel.

In our case study, we applied a model-based test generation approach to test the validity of the implemented handover functionality of an RBC. We used both the introduced prototype as well as the proposed process of refining test artifacts generated by MBT, i.e., creating the traceability from the automatically generated tests to requirements. Our working procedure consists of four steps:

1. modeling the requirements with our extended SysML,
2. modeling the test basis, i.e., the UML state machine,
3. generating the test cases based on a coverage criterion (here "round trip path") at model level, and
4. evaluating the generated test cases.

In the first step, we modeled the requirements with our extended SysML. The functionality of the system under test (SUT) is defined in two chapters of the Subset-039: Chapter four lists basic tasks of both roles in the handing-over transactions and chapter five defines complex requirements, among which are repeatings, overlapping, and concurrency of transactions. We used the paragraphs given in chapter four of the Subset-039 to model the atomic requirement with sequences and the paragraphs of chapter five to deduce the complex models of requirements with our extended SysML requirements diagram.

For the second step, the state-based view of the functionality given in Subset-039 is reused as our test basis model. We created a corresponding state machine with two orthogonal regions to model the concurrent behavior of the SUT in the role of a handing-over and an accepting RBC. A lot of RBC functionality that is not in the scope of RBC/RBC Handover was deported into the system environment, e.g., the capability to discover that RBC/RBC handover is necessary for a train. As defined in Subset-039, we modeled two states (idle and HOV respectively ACC). The resulting state machine has 42 transitions equally spread over both regions and only two states per region, which results in a very high number of possible concrete interaction sequences. The firing of the transitions was modeled in a standard UML way, i.e., a transition can fire if the corresponding event is triggered and the guard is true. Triggering events can have data values as parameters that can be processed by the transitions of the state machine, e.g., by evaluating event parameters in their guards or assigning them to attributes.

In the third step, we use a Petri net-based test generator [7] for model-based test generation. With this test generator, strong coverage criteria (up to "round trip path" coverage) on the model level can be achieved. It uses standard Petri net methods combined with constraint programming techniques for representing the behavior of a model, meaning all possible execution paths are coded in a easy understandable way (e.g. reachability graph, unfolding of a Petri net). If necessary, the test generation algorithm can operate on the whole state space achieving high coverage. Additional verification tasks like the detection of deadlock, lifelock, or reachability are supported with this method. For the modeling the Petri net dialect SPENAT (safe Petri net with attributes) is used. With a SPENAT one can use Petri net modeling with syntactical constructs like trigger events with parameters, guards, actions, and attributes. With these properties the mapping of a UML state machine to a SPENAT is possible and can be implemented in a intuitive way [7]. The result of the model-based test generation with the mentioned strong coverage criteria are 2812 test cases with around 20733 sent messages covering both roles of the handover functionality.

In the fourth step, we were now able to evaluate and validate the generated tests. Therefore, our prototype has two outputs of which the report is of prior interest. In the report, test cases without reference to requirements are listed as well as requirements that could not be tested with the set of test cases. Our case study contains 34 requirements, 25 of which are simple requirements and 9 of which are complex requirements. While almost all test cases are reported to cover at least one atomic requirement (which is due to all transaction processes are opened by an atomic requirement) a lot of test cases do not cover any of the complex requirements. Only 209 out of 2819 test cases contained traces to complex requirements. Also interesting, 7 test cases did not even cover an atomic requirement. Furthermore, the major problem of our generated test cases is indicated by the report that some of the complex requirements are not covered by any test case at all. Among these complex requirements are those that describe all the mandatory and optional atomic requirements to build up a successful handover transaction. 5 out of 9 complex requirements were not covered by any test case. Although we seemed to have a correct state machine according to Subset-039 and to our own review and a correct MBT tool, test case generation failed to cover complex requirements. The reason of that problem is not model-based testing but the insufficient way our test basis model is derived from requirements and processed by test generators, afterwards. The efficiency of test generators leads to test processes that are incomplete from the functional notion that is anchored in the requirements model. Thus, test base models must be created in a way that is functional correct but also responsive to the applied methodology of test case generation. In order to increase the quality of the generated test cases in our case study, two solutions seem appropriate: The used model with the low number of states must be better programmed by constraints to lead the test case generation into better functional processes. Another solution would be to increase the number of states to define the order of transitions like in [15]. As a consequence, we focus our future work on the ability of detecting incompleteness in generated test cases and on improving the system model that is used for test generation. With our methodology, test basis modeling can be done in a cycle process that leads to satisfying test cases much faster than with manual analyzes of the test artifacts.

7 Conclusion, Discussion, and Future Work

In this paper, we showed a new way of automatically creating traceability from test cases to requirements. We described the approach, the corresponding prototype implementation, and a case study with a concrete example from the railway domain. We defined an extension of SysML requirement diagrams that was used used for automatic traceability. Our way of automatically tracing tests to requirements is based on pattern matching of concrete test sequence to abstract patterns defined in requirements. The independence of our tracing approach with respect to the test design process has the important advantage that it can be applied to a majority of test generation approaches. In other words: Using our approach, a majority of automatic test design techniques can be seamlessly integrated into tool chains that are used for the development of safety-critical systems. Both the refinement report and gathered traceability matrix bring benefit to software testing: First, evaluating and validating test cases, i.e., to prove that requirements are covered is more easy and quicker than by manual analyzes. Second, the gathered traces can be used for better system verification and validation, for checking requirements satisfaction on system level, and for debugging after test execution.

As a major result of our case study, many atomic requirements were covered by the automatically generated test cases while several complex requirements were not - although the structural coverage criterion of the model level was satisfied. We identified deficiencies of the test basis model with respect to the used test case design principle and coverage criterion as the reason for that incomplete coverage. In future work, a test design process should include modeling, test case generation, and above all requirements evaluation.

Although the first results are promising, there is room for discussion. For instance, not necessarily all requirements can be expressed as behavioral models such as sequence diagrams. While this is obvious for untestable or static requirements, many requirements can be expressed as behavioral models. This is backed up by the experiences described in our case study. SysML requirements diagrams also allow for linking untestable requirements to our complex and atomic requirements. Furthermore, the case study shows that model-based test generation was not able to cover all complex requirements. The most important question is for the reason of this. We think that structural coverage criteria are not the best means to cover functional requirements [15] and model-based test generation has to be adapted accordingly. Finally, our approach requires to create two models (a test basis model and a requirements models), which causes additional effort for maintenance and change management and, thus, reduces acceptance. Thus, our approach could suffer from redundancy, though requirements models are helpful for other steps in software development. We plan to resolve this problem in future work by deriving all models from the requirements models.

For future work, we identified the following tasks. First, we found out that coverage of complex requirements with a high number of possible interactions of atomic requirements is hard to enforce. For instance, one can state that two sequences are independent, i.e., they can be run in any order. However, the tester is definitely not going to run all of these possible interactions. As a consequence, we thought of defining coverage criteria like *cover at least one execution order with n switches of the active test sequence*. We plan to develop an inverse approach of the presented methodology that is able to create test cases out of the extended SysML requirements model instead of state-based test generation. With this approach, we could reduce the effort of modeling because separated (and partly redundant) test basis models are not necessary anymore. Second, we want to extend our approach to work with other test generators, e.g., commercial ones such as Conformiq Designer or the MBTSuite. Finally, we plan to make the developed prototype open source and to integrate it in a tool chain that allows exchanging single components such as test generators.

References

1. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.
2. Boris Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
3. Conformiq. Conformiq Designer. http://www.conformiq.com/.
4. Mario Friske and Holger Schlingloff. Abdeckungskriterien in der modellbasierten Testfallgenerierung: Stand der Technik und Perspektiven. *Workshop Model-Based Development of Embedded Systems (MBEES)*, 2006.
5. Thilo Girlich. Anforderungsbezogene Priorisierung und Bewertung modellbasiert generierter Testfälle, 2012.
6. imbus. Testbench. http://www.imbus.de/produkte/imbus-testbench/.
7. Jan Krause. *Testfallgenerierung aus modellbasierten Systemspezifikationen auf der Basis von Petrinetzentfaltungen*. PhD thesis, Institut für Automation und Kommunikation Magdeburg e.V. ifak, 2012.
8. Object Management Group. Unified Modeling Language (UML), version 2.4. http://www.uml.org/, 2011.
9. Object Management Group. Systems Modeling Language (SysML), version 1.3. http://www.omgsysml.org/, 2012.
10. sepp.med. MBTSuite. http://www.seppmed.com/.
11. Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.
12. Stephan Weißleder and Dehla Sokenou. ConSequence - Model-Based Testing With State Machines and Concatenated Sequence Diagrams. *Gesellschaft für Informatik, 32nd TAV meeting*, 2012.
13. UNISIG. Subset-039: Fis for the rbc/rbc handover, 2009.
14. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
15. Stephan Weißleder. Simulated Satisfaction of Coverage Criteria on UML State Machines. In *International Conference on Software Testing, Verification, and Validation (ICST)*, April 2010.