Silvio Marcovic, BSc

Integrating an External Test-Case Generator into a Property-Based Testing Tool for Testing Web-Services

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Aichernig, Bernhard, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute of Software Technology (IST)
Graz University of Technology, A-8010 Graz, Austria

Co-Supervisor
Schumi, Richard, Dipl.-Ing. BSc Ing.

Graz, September 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

25.09.2017

Date

[illegible]

Signature

Abstract

Software testing consumes much of the total development effort, with manual testing still being a considerable part of it. To lower the costs and to improve the efficiency of testing, a great amount of research was conducted in the area of automated test-case generation. This thesis focuses on developing an interface to combine different automated testing strategies.

There is an abundance of model-based testing tools available. The manual creation of models and their integration into the software development process are still critical issues. This work presents a test-sequence generation approach for applications driven by business rule engines. The application's behavior is stored in these rules in an abstract manner. We use a property-based testing (PBT) tool, FsCheck, to generate test sequences. A property is a high-level specification of the behavior of a code unit. We derive models and test-data generators from the rule engines to form properties of a system under test. We define these properties in C# and verify if the properties hold for our system or not.

Furthermore, we work on exploiting the PBT features and combining them with external test-sequence generators. An interface for FsCheck is presented that allows us to integrate other generation strategies such as mutation-based testing into it. We aim to give the tester more control over the production of meaningful operation sequences for test cases. By integrating an external test-case generator into a PBT tool, we can create test cases that follow certain coverage criteria. This is shown by combining the model-based mutation tool, MoMuT, as an external generator with FsCheck to create model-based mutation tests.

Integrating MoMuT allows us to reduce the test execution time, as we do not need to create a large number of random tests to cover certain model aspects. We demonstrate our interface with a simple example of an external generator based on regular expressions and, finally, conduct a case study, where we integrate MoMuT into FsCheck, discuss and evaluate the approach.

Keywords: Model-Based Testing, Property-Based Testing, Test-Case Generation, Mutation-Based Testing, MoMuT, FsCheck, Web-Services, Business-Rule Models

Kurzfassung

Das Software-Testen ist ein großer Anteil des gesamten Entwicklungsaufwandes, wobei das manuelle Testen noch immer eine entscheidende Rolle spielt. Um die Kosten zu reduzieren und die Effizienz zu steigern, wurde im Bereich der Testautomatisierung viel Aufwand in die Forschung investiert. Ziel dieser Arbeit ist es, eine Schnittstelle zu entwickeln, um verschiedene Testautomatisierungsmethoden zu verbinden.

Es existiert bereits eine Menge an modelbasierten Testwerkzeugen. Jedoch ist die Generierung von Modellen und deren Integration nach wie vor ein kritisches Thema des Softwareentwicklungsprozesses. Diese Arbeit präsentiert einen Prozess zur Testfallgenerierung für Applikationen, die von Business-Rule-Models angetrieben werden. Das Applikationsverhalten ist in diesen Regeln in abstrakter Art gespeichert. Wir verwenden FsCheck, ein Property-Based-Testing Tool, um Testsequenzen zu generieren. Eine Property ist eine Spezifikation des Verhaltens einer Programmeinheit. Wir leiten Modelle und Testdatengeneratoren von den Business-Rules ab, um Properties für ein zu testendes System zu erzeugen. Diese Properties sind in C# definiert und es wird überprüft, ob diese für das System gelten oder nicht.

Zusätzlich haben wir Teile aus dem Property-Based-Testing extrahiert, um diese mit anderen Testsequenzgeneratoren zu verbinden. Es wird eine Schnittstelle für FsCheck präsentiert, welche es erlaubt, andere Testgenerierungsmethoden wie mutationsbasiertes Testen zu integrieren. Wir versuchen dem Tester mehr Kontrolle über den Generierungsprozess zu geben, um sinnvolle Sequenzen für Testfälle zu erzeugen. Durch die Integration eines externen Testfallgenerators in ein Property-Based-Testing Tool ist es uns möglich Testfälle zu generieren, welche bestimme Testabdeckungskriterien erfüllen. Dies wird durch die Integration von MoMuT, einem model- und mutationsbasierten Testwerkzeug, als externer Generator in FsCheck gezeigt. Mit dieser Methode ist es möglich, modelbasierte Mutationstests zu erzeugen und die Vorteile von Property-Based-Tests zu nutzen.

Durch die Integration von MoMuT konnten wir die Testlaufzeit verringern, da wir weniger Tests gegenüber dem Zufallstesten benötigen, um gewisse Modelaspekte abzudecken. Wir beweisen die Durchführbarkeit unseres Prozesses durch das Beispiel eines Regular-Expression-Based-Generators. Schlussendlich haben wir eine Fallstudie durchgeführt, welche MoMuT in FsCheck integriert, in der wir die Methode dieser Arbeit besprechen und evaluieren.

Schlagworte: Modelbasiertes Testen, Property-Based-Testing, Testfallgenerierung, Mutationsbasiertes Testen, MoMuT, FsCheck, Web-Service, Business-Rule-Models

Acknowledgements

I would like to thank all people that supported me during my studies and especially during the time I was working on my thesis. I express my deep gratitude towards my supervisor Bernhard K. Aichernig who raised my interest and helped me learn various new software-testing techniques. This interest is what ultimately led me to write this thesis. During my work on the thesis, he provided me with guidance whenever needed.

I want to thank Richard Schumi who helped me a lot with progressing on the practical parts of my thesis and also for the countless reviews on the written part of it. This thesis also benefited a lot from his research prior to and alongside this thesis.

Furthermore, I want to thank Martin Tappler with whom I had various discussions about this thesis and who pointed me into correct directions whenever I was straying. Especially his help for MoMuT related topics was beneficial.

I am thanking the employees of AVL List GmbH who enabled me to validate the developed technique on an industry case-study. This helped me gain confidence in the usefulness of this work.

Lastly, I would like to thank Alexandra Ospanova for her mental support, which helped keep my motivation up through all the up and downs of writing. Additionally, I would also like to thank her for her proofreading, which has profoundly improved the language of this thesis.

Silvio Marcovic
Prague, Czech Republic, 03.09.2017

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Motivation

Software testing is an important aspect of the software-development process. Utting and Legeard [107] state that software testing generally consumes between 30 and 60 percent of the overall development effort. Software testing costs are often 50 percent or more of total development costs [11, 87]. Fagan [40] point out that less than 50 percent of the effort for operations labeled as testing is actually spent on verifying that the product meets the requirements. Most of the effort is consumed doing defect rework or in other words fixing mistakes that were discovered during testing. This indicates that the testing expenses are distorted from reality. Nevertheless, developing a suitable test framework, finding and fixing mistakes are all pressing topics in software development.

As an example, the development of Windows Server 2003 had a team of 4400 software engineers, 2000 of them formed the development team while 2400 were in the test team. With around 50 percent spent on development and 50 percent spent on testing, software development is a unique discipline. In traditional manufacturing industries development accounts for more than 99 percent of the costs. The lack of automation and hence the absence of process control makes it difficult to control product quality [78].

It is evident that testing software and ensuring its quality consumes a lot of time and money, and testing costs are still growing. In order to tackle these problems testing strategies have to become more elaborate.

One of the most critical challenges in testing is the generation of test cases. A great amount of research was conducted over the past few decades. Far-reaching automation is a way to ensure software quality in the evermore complex software [20]. There are several different techniques for automated test-case generation. According to Anand et al. [11] and Keyvanpour et al. [64] these techniques include, among others:

- model-based test-case generation;
- symbolic execution and structural coverage testing;
- search-based testing;
- random testing.

The approach for automated test-case generation presented here utilizes two different tools, which support model-based test (MBT) case generation: FsCheck [9] and MoMuT [67].

FsCheck is a property-based testing (PBT) tool for .NET inspired by QuickCheck [52]. The tool generates random inputs and tests properties [34]. A property is a high-level specification of behavior of a code unit. A variety of PBT tools also support stateful testing with their dedicated frameworks, as does FsCheck. The property of a state machine can be expressed in following way. The state of the system under test (SUT) conforms to the state of an abstract model after a sequence of operations is executed on both, the SUT and the model. MBT and especially FsCheck are further explained in the Chapters 2 and 3.

MoMuT, which stands for MOdel-based MUtation Testing, applies the idea of mutation testing where faults are inserted into the source code and the effectiveness of a test suite is measured by how many of the mutated programs are detected. Instead of inserting faults into the source code MoMuT inserts them into the model of the SUT instead and tries to generate test cases revealing those faults [67].

Our approach for test-case generation focuses on rule-engine driven web-service application. Many web-services store their configuration in XML files. Some even store their business logic such as access rules and workflow details in XML business-rule models (BRM) [91, 92].

These XML definitions can be seen as abstract specifications of the service behavior and serve us as input for our MBT tools. We transform these specifications into an extended finite state machine (EFSM). An EFSM model is a generalization of the traditional state machine model. It is extended by a set of variables and a set of trigger conditions, which when enabled fire a transition and alter the state of the variables. They are explained in further detail in Chapter 5.1. The EFSM model serves as an intermediate format from which we can continue the translation process to prepare the input for either MoMuT or FsCheck. MoMuT is then executed to find transition sequences for the SUT. FsCheck takes those sequences as input and randomly generates the data needed to perform those transitions.

This approach is an extension based on the prior research of Aichernig and Schumi [9]. In the prior approach the BRMs were translated into EFSMs and further transformed to serve as input for FsCheck to generate test sequences with random data. In the approach of the thesis we extended the initial idea by using an external test-case generator to gain better control of the test-case generation. We aim to reduce the length and therefore the complexity of generated test sequences while simultaneously trying to keep the model coverage high. This would allow us to reduce the test-execution time. How this is accomplished is further explained in the next sections.

This thesis focuses on MBT approaches. It builds upon existing tools and provides methodology to combine those. We aim to use existing test-generation strategies to combine them into an elaborate testing strategy which helps us reduce the testing time costs. We evaluate our approach by embedding MoMuT as an external test case-generator into a PBT tool FsCheck to test a web-service application.

1.2 Problem Statement and Solution

MBT is an effective approach to test software. However, there are still open issues related to MBT. One of the most difficult parts is the creation of models. Creating models manually is a complicated task, and two model developers writing a model for an application will probably come up with different models which will most likely result in different test suits [44]. Another issue related to models is that they are often defined exclusively designed for an MBT approach and are not integrated with the software-development process [39].

The first goal of this master thesis is to tackle the stated problems. The model in our testing approach is derived from business rules which are also used in the implementation of the system. With the presented approach we focus on how to derive models automatically, and hence do not need to create models manually. While the models are defined for MBT we derive them from business rules and therefore have them integrated into the software-development process. If the system and its rules change, our model representation changes with them. Of course, this still means that the testing framework needs to be maintained in case the rule engine syntax or semantic changes.

We apply PBT to generate test sequences and data to execute tests on the system and our models. PBT has gained a lot of attention and has become the dominant testing tool in the Haskell community [53]. With the state machine libraries for MBT many large systems were tested [54]. While some of the studies resort to other tools for the testing approach, no research was found on the integration of other test-case generation tools into an existing PBT tool. There were no measurements of test coverage in the early versions of QuickCheck, which was one of the tool's major limits [34]. While newer versions of QuickCheck and QuickCheck inspired tools now support means to investigate the distribution of test data we are not aware of any PBT tools that generate test sequences based on coverage criteria.

The second goal of this thesis was to further improve the approach to have fewer test cases and thereby reducing the test execution time. We also aimed to gain better control on how test sequences are generated. This can be accomplished by integrating other (external) test-case generation strategies into a PBT tool. With MoMuT as an external generator and the implementation of observer automata we are able to create test cases that are guaranteed to cover aspects of the model and still exploit the benefits of PBT. Integrating MoMuT is only one way of incorporating a different test-case generation strategy into a PBT tool. The solution we present in this work is generic and not only solves the problem of generating test cases that cover model aspects. The solution can also be applied to tackle additional test goals such as other coverage criteria, minimizing test generation and execution costs.

The presented approach was evaluated in an industrial case study. A web-service application was provided by the industrial project partner of this thesis, AVL List GmbH (AVL)[1]. Their tool is used for management of resources, data, and workflows and is further divided into several modules. The system is further explained in Chapter7. The results in the study are mainly focused around two of the bigger modules. However, in the progress of the thesis other modules were also analyzed to help develop the presented approach. The results of the study suggest that we were able to reach the set goals for this thesis. For the tested modules, we managed to generate test cases that were able to find faults in the system with shorter test cases and less execution time as compared to not using MoMuT as an external generator. However, the case study also shows the limitations of our approach, namely, the expensive test-case generation time.

1.3 Related Publication

Prior to the work of this thesis, a paper has been written by Aichernig and Schumi [9]. It covers the base on which this thesis is built upon. As a result, some techniques and concepts discussed in this thesis have already been discussed in a similar form beforehand. Especially parts of Chapter 5 are based on this prior research.

The author of this thesis has been a co-author for a publication related to PBT [7]. This paper has been presented at the **13th Workshop on Advances in Model Based Testing (A-MOST 2017)**[2] in Tokyo, Japan on 17 March 2016. It contains important key concepts that are presented in this thesis.

These two publication serve as a clear distinction between prior research an contribution of the author. Therefore, they will be discussed here shortly.

1.3.1 Prior Research

Property-Based Testing of Web Services by Deriving Properties from Business-Rule Models
This paper discusses how to use a PBT tool to automatically derive generators and models from rule-engine driven web-service applications. XML BRMs are parsed into EFSMs, which serve as a intermediate format between the original models and the PBT tool. An EFSM is a very general concept and it is easy to infer model-based tests from it, which makes it a convenient format for in the translation process. An EFSM model is further translated into the specification and command needed for FsCheck. This enables stateful testing with random input data. The SUT is tested by executing those random sequences and searching for a difference between the specification's and the SUT's state. The test case fails if a difference was found or if an exception occurred during the run; otherwise, the test case passes. An overview of the process is illustrated in Figure 1.1.

[1] `https://ww.avl.com`
[2] `http://a-most17.zen-tools.com`

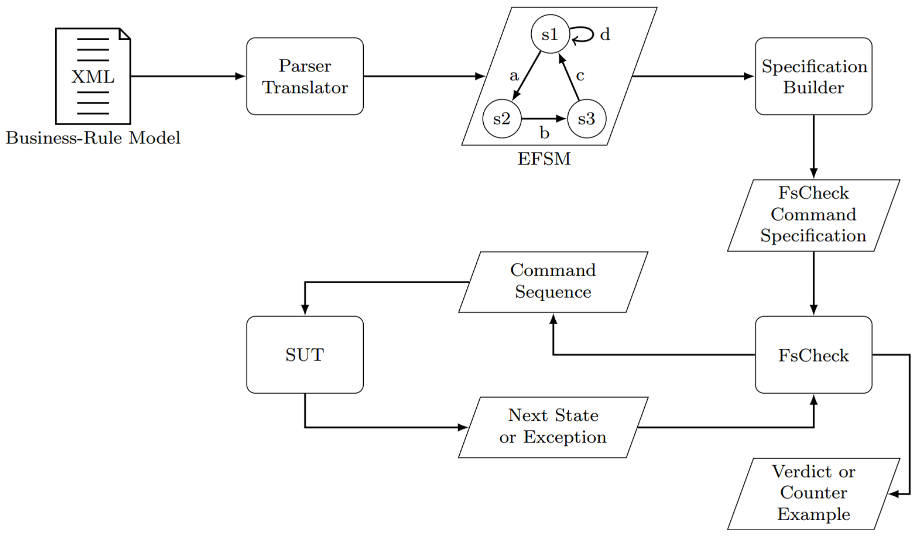

Figure 1.1: Overview of the steps for the FsCheck command sequence generation for business-rule models.

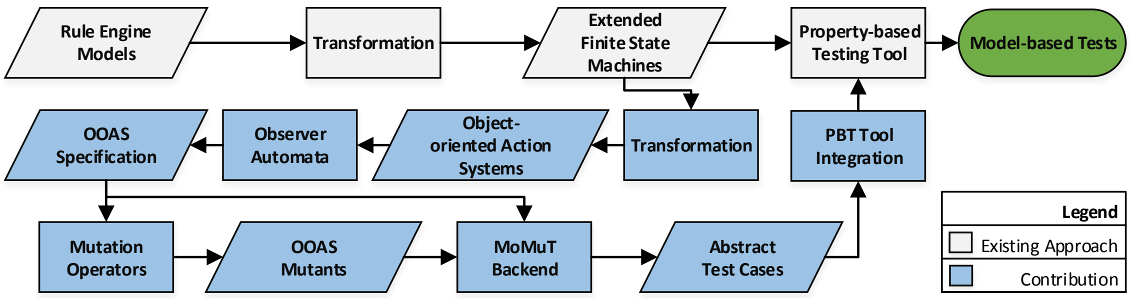

Figure 1.2: Overview of the steps for the integration of an external test-case generator.

The method has been applied to the BRMs of an industrial web-service application. The author of this thesis has helped to test the modules and to discover faults in the SUT.

1.3.2 Contribution of the Author

Property-Based Testing with External Test-Case Generators This paper focuses on combining the approach from the previous research with external test-case generator. A generic approach is presented on how to integrate external tools with PBT tools, by giving the example of applying MoMuT as an external test-case generator and integrating the produced test sequences into the PBT tool FsCheck. With having more control over the test-sequence generation it is possible to produce meaningful operation sequences for test cases. With this approach, we are able to generate test-cases that follow certain coverage criteria. This allows us to reduce the overall test-execution time because coverage controlled tests are in general shorter than random tests. An overview of the approach is presented in Figure 1.2. In addition the figure also shows a clear distinction between prior research and contribution of the author of this thesis.

A case study on the same system as in the prior research was performed to validate the new approach. The case study was further extended by testing additional modules.

1.4 Research Project: TRUCONF

This work is part of a research project called TRUCONF[3]. This section will contain a summary and goals of the project and explain how this thesis contributes to the project.

TRUCONF stands for trust via cost function-driven model-based test-case generation for non-functional properties of systems of systems and is a research project which is funded by the Austrian Research Promotion Agency (FFG). The project has a planned duration of three years which started in November 2014 and will end in January 2018. There are two research and one industrial partner contributing to the project: The Austrian Institute of Technology (AIT)[4], the Institute for Software Technology (IST) at the Graz University of Technology and the industrial partner AVL.

1.4.1 Project Summary

The project is driven by a use case of AVL. They developed a system, called AVL Testfactory Management Suite (TFMS) [5], where multiple test devices are combined into test beds per customer needs. The automation system they developed controls the test beds and the plugged-in devices. The tool is divided in multiple models for management of resources, data and workflows and interacts with test-automation-systems, project-management systems and unit-under-test databases. The system is explained in more detail in Chapter 7.

Currently, testing of the automation system is done manually, which is a time-consuming process that also cannot guarantee coverage of worst-case scenarios. In a lot of cases issues arise from shared use of limited resources. For example, bandwidth and memory. For every new software release a lot of manual effort is invested. Therefore, they try to shift from manual to automated testing by applying and extending MBT techniques.

Since sharing of limited resources was identified as one of the causes of issues, high consumption of resources is a good indicator to reach erroneous scenarios. An approach will be developed which extends an existing MBT generator with support for resource-consumption testing. With this approach they are moving from testing of functional properties to testing non-functional properties. The project will develop tool support to automatically derive resource consumption data from a SUT and connect to a given behavior model. The main goal of the project is to create test-case generation for non-functional requirements and further use the data from test-case execution to tune the models with the help of cost-function refinement.

1.4.2 Contribution to the Project

This section will briefly explain how this thesis contributes to the TRUCONF-project. In this thesis we automatically derive models for systems by combining MBT and PBT approaches. We further extend these models to reduce the execution time of test sequences by integration MoMuT as an external test-case generator.

This approach contributes to the first goal of the project. While the approach of this thesis does not test non-functional requirements as stated as a goal, it provides the automatically-derived models and also some of the needed tool support. The second goal of the project is not covered by any content of this thesis. In addition, the approach of this thesis was evaluated as part of a case study. In the case study we tested parts of AVL's systems and were able to find potential issues in them. They are further explained in Chapter 7. This thesis extends the test coverage and shows possible limitations of the SUT.

[3]`http://truconf.ist.tugraz.at`
[4]`https://www.ait.ac.at`
[5]`https://www.avl.com/-/avl-testfactory-management-suite-tfms`

1.5 Structure of this Thesis

The rest of this master's thesis is structured as follows:

Chapter 2 gives a broad overview of the theory and technologies on which this thesis will build upon. It contains an introduction of the applied testing strategies and on the theory that was needed to perform and evaluate the case study.

Chapter 3 introduces FsCheck, a PBT tool written in F#, which is easily accessible from .NET code. The presented approach of this thesis is implemented into FsCheck and it is therefore important to understand how this specific PBT tool operates. Mainly, its features of generating random values and forming test sequences to perform MBT are explained.

Chapter 4 describes a model-based mutation-testing tool called MoMuT. MoMuT was selected as the external test-case generator for our case study. The chapter aims to provide knowledge about the language and the test generation process of MoMuT. Chapter 3 and Chapter 4 provide a good basis to understand the main contributions of this thesis.

Chapter 5 and Chapter 6 form the main contribution of the thesis. Chapter 5 explains how test cases can be automatically derived from rule-engine models. It is shown how a REM can be translated into an appropriate format for a PBT tool to perform MBT. The focus of the chapter lies on the translation steps necessary and on the explicit implementation steps required to utilize FsCheck for MBT.

Chapter 6 focuses on extending the methods presented in the previous chapter by implementing external generators. An interface is developed to implement the external generators into PBT tools. Two examples of external generators are given to show the usage of the interface. The first external generator is a simple regular expression-based generator and illustrates the use of the interface. The second external generator is MoMuT. It is shown how MoMuT sequences are executed within FsCheck.

Chapter 7 contains a case study that evaluates the methods from Chapter 5 and compares them to the method with MoMuT as an external generator. It provides evidence on the generation characteristics and discusses the benefits and drawbacks of both strategies.

Chapter 8 gives a summary and an overview of related work. PBT of web-services, as well as a combination of other tools with PBT, will be discussed. Finally, a discussion of findings and future work for implementing external test-case generators concludes this thesis.

2 Background

This chapter provides theory and technologies on which this work is built upon. In Sections 2.0.1, 2.1 and 2.2 an overview of the relevant testing strategies is given. Then, in Section 2.3 business-rule models (BRMs) and related terms are explained. The system analyzed in the case study is built on them. Finally, Section 2.4 explains and discusses coverage criteria, which are widespread metrics to evaluate test suites.

2.0.1 Model-Based Testing

Instead of writing test cases manually, the designer writes an abstract model. The test cases are then generated from the model. Model-based testing (MBT) can be applied to different scales of the SUT ranging from smaller units up to the whole system. Binder et al. [22] surveyed a hundred MBT users. The survey revealed that most people apply MBT for system testing (almost 80 percent) and integration testing (50 percent). Dias Neto et al. [39] made a survey of MBT approaches. 66 percent of all approaches were system level approaches while integration testing made up for 22 percent.

The MBT process can be divided into the following steps, as shown in Figure 2.1 [107].

The first step of MBT is to write a model. The model is an abstract representation of the SUT. It should be smaller and simpler than the SUT. It serves as a specification and should focus on the key aspects that the designer wants to test [74, 107].

The next step is generating tests with adequate criteria. Since possible input grows very large even with small models, appropriate selection criteria have to be chosen. There exists a wide range of coverage criteria. Data-flow and control-flow criteria have been adopted from code-based testing. Transition systems describe potential behavior of systems and consist of states and transitions, which may be labeled. Labeled transition systems(LTSs) are explained in more detail in Chapter 4. Transition-based notation makes it possible to apply graph-based coverage criteria such as node or edge coverage [108].

If the coverage criteria are not satisfying, additional abstract test cases can be added manually. The set of tests is then transformed into concrete tests, which are then run on the SUT. Finally, the results of the test run should be analyzed. It is important to note that *online* tools usually merge the generation of abstract tests with the transformation and execution process while *offline* tools do this step-by-step [107].

In order to analyze the results of the test run we can examine the behavior of a system with an oracle. Given an input for an SUT the oracle distinguishes between the correct behavior and potentially incorrect behavior of the system. There are different techniques for oracle automation. It is possible to create the oracle from the model. If the oracle automation is not adequate the human serves as an oracle. He may be aware of specifications and expectations that are hard to quantify. Especially during the first executions of a test set, errors arise from the adaptor code or the model. Certainly, an oracle derived from the model can only be adequate if the model itself is a correct representation of the SUT [19, 107].

2.0.2 Tools

In this section, a few tools are listed that help create model-based tests. AETG [35] is a tool for combinatorial testing. It contains an algorithm to achieve pairwise coverage, which results in a huge reduction of test cases. The oracle has to be provided manually and it supports offline tests. SpecExplorer [109] is a tool developed within Microsoft Research which supports state-based models and both, online and offline testing.

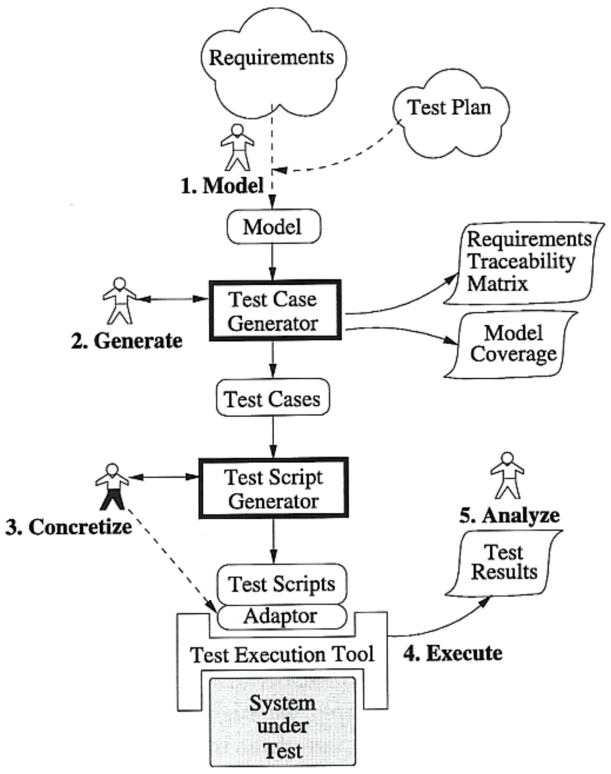

Figure 2.1: The model-based testing process (testing tools are in boxes with very bold lines).

UPPAAL TRON is an MBT tool for real-time systems, which uses UPPAAL as its model checker. UPPAL was developed at the University of **Upp**sala and **Aal**borg, hence the name. The tool checks if the SUT conforms to a specified model performing input-output conformance (ioco) checks while taking time constraints into account. Expressing the specification as extended finite state machines (EFSM) with a set of real-valued clock-variables allows the tool to keep track of time-related behavior [49, 69]. Ioco will be explained in Chapter 4 and EFSMs in Chapter 5 since the parts of this thesis that rely on the background are introduced there.

Other tools are Torx by Tretmans and Brinksma [106], which also uses ioco to select their test cases, and Conformiq Qtronic by Huima [55], which uses symbolic execution to reduce the model space that needs to be executed. Those tools are not further explained since their details are not directly relevant to the content of this thesis.

2.1 Mutation Testing

The idea of program mutation has a long history. It goes back to 1971 when it was proposed by Lipton [73]. In mutation testing, a set of elementary mutation transformations are applied to introduce faults of certain types into the program. The goal is to construct a set of tests which are able to distinguish between the original program (SUT) and non-equivalent mutated programs [51].

Note that we apply a precise meaning to the term fault and distinguish between other terms that are commonly used in similar context. In the following, we distinguish between the terms *Mistake*, *Fault*, *Error* and *Failure* and use them as defined by Radatz et al. [88].

Error. The difference between a computed, observed, or measured value or condition and

the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.
Fault. An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program.
Failure. An incorrect result. For example, a computed result of 12 when the correct result is 10.
Mistake. A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator.

The general idea is that the faults generated by mutation testing represent mistakes programmers often make. *Competent programmers* do not create random programs. Through many iterations, they create programs that are close to what they intend the program to look like. Test data that distinguish all programs differing from a correct one by only simple errors are so sensitive that they also implicitly distinguish ones with more complex errors. This is called *the coupling effect* [37]. Therefore, mutation testing usually relies on simple mutants only.

The adequacy of a test suite can be measured by a mutation score. The score is calculated by dividing the number of mutants killed by the number of non-equivalent mutants [36]. A mutant is killed if the test suite identifies a different behavior compared to the original program.

2.1.1 Unresolved Problems

While mutation testing is an effective way to evaluate a test suite, it has a number of problems. One is the high computational cost of executing the mutants against a test set. A second one is the equivalent mutant problem.

The most apparent way to reduce the costs of execution is to reduce the number of mutants. Raunak et al. [90] show that with off-by-one loop mutation the costs to traditional mutants can be reduced while maintaining an adequate test suite. They produced 89.15 percent fewer mutants while the mutation scores had a correlation coefficient of 92.28 percent. This type of cost reduction is called constrained mutation. There are other methods such as randomly selecting a subset of mutants [111].

Equivalent mutants are mutants that have no functional effect on the program and thus cannot be killed [84]. Examples are mutations in unreachable code and mutating code like $a = \neg b$ to $a \neq b$. Methods to solve the problem include compiler optimization techniques, program slicing, and many others. A comprehensive literature review was done by Madeyski et al. [76].

The comprehensive survey of Jia and Harman [60] found evidence that the literature on mutation testing is growing. They state that even though mutation testing is still suffering from several problems, it is reaching a maturity not previously witnessed in the field.

2.1.2 Tools

A few mutation tools are discussed in this section. The first mutation tool was developed by Acree et al. [1] in 1980 at the Yale University. They presented a mutation system and mutation operators for Fortran and Cobol.

There are also mutation-testing tools for higher level languages. A set of object-oriented mutation-operators for C# was introduced and implemented into the tool CREAM (Creator of Mutants) [38]. Mutation operators have also been proposed for SQL, Java, PHP and many other high-level languages [30, 65, 98].

Since mutation testing became a popular technique for testing software there exists a vast variety of tools that are not covered in this section. Jia and Harman [60] conducted a survey of the development of mutation testing and explain a variety of tools. The interested reader may refer to this source for information on additional tools.

2.1.3 Model-Based Mutation Testing

As mentioned in the introduction, the tool used in this research is MoMuT. MoMuT combines model-based testing, where an SUT is tested for conformance with a model specifying the intended behavior, and mutation testing, which intentionally alters the model of the system to ensure a fault-oriented testing strategy. Compared to tradition mutation testing, the mutations are not performed on the source code of the SUT but on the model. With mutating on a high level of abstraction black box testing is possible since the internal structure of the SUT is not needed to design a model.

MoMuT is able to generate its tests from unified modeling language(UML) state machines. UML is a widespread modeling language that helps to visualize a system's design in a normalized way. The UML state machines are translated into action systems and further interpreted as LTSs. So far MoMuT only supports offline testing [67, 66]. The tool is explained in more detail in Chapter 4.

2.2 Property-Based Testing

In the past, several researchers used the term property-based testing (PBT) to describe their testing techniques. Some of the techniques aim to produce testing criteria based on abstract models [42, 86]. In 1999 the influential PBT tool QuickCheck was released [34]. It is written in Haskell and quickly became established in the testing community. Nowadays the term PBT is usually understood as a synonym for QuickCheck and re-implementations of QuickCheck. Tools from the QuickCheck family and previously reported methods have in common that they derive test cases from specifications of a code unit. This thesis will use the term PBT to describe the techniques of the QuickCheck family.

A property is a high-level specification of the behavior of a code unit. Properties are checked by executing the code unit with a range of data points. If no execution violates the property we assume the property holds. If a single contradiction is found the property does not hold. A common example of a property is that if the reverse is applied twice to a list it will be equal to the original list. This is shown in Figure 2.2 [110]. More general, when a function f has an inverse f^{-1} then

$$\forall x \in X \left(f^{-1}\left(f\left(x\right)\right) = x\right)$$

where X is the set of possible inputs. In our example, the inverse of the reverse is conveniently reverse itself. These properties are tested on random input which is supplied by test-data generators. Let

$$C = (c_1, c_2, ..., c_n) : c_i \in Y$$

be a collection where the elements are elements of a set Y such as the set of real numbers $\mathbb{R}$. Then we can write a property in the following manner:

$$\forall C : C = List.reverse(List.reverse(C))$$

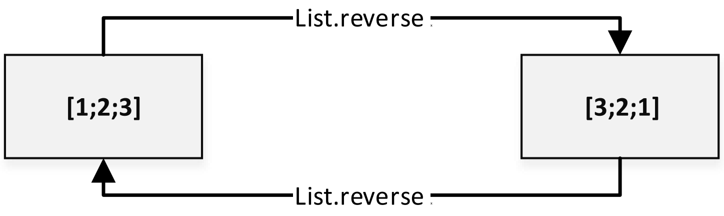

Figure 2.2: The reverse of the reverse is equal to the original list.

A generator generates the data points for a certain type of input. In the above example, the generator will create collections that contain elements of the set Y. Some common generators are already implemented in the framework. However, a user can create his own generator. QuickCheck then generates test data through the generator and tries to find a counterexample for the property. If no counterexample is found for a sufficiently large number of tests we assume that the property holds.

We will now try to define a second useful property for our *List.reverse* function. For this, we introduce two functions. $List.insert(C, i, n)$, which will add an item n at a specific index i of the list and $List.count(C)$ which will return us the number of items currently in the list. The next property will make use of the fact that certain functions can be combined in a different order but still lead to the same final result. For example, taking a step to the right and then to the front is the same as taking a step to the front and then to the right. We will compare the equality of the resulting lists between the following two combinations of functions.

1. Inserting an item n into the list at the position i and reversing the list.
2. Reversing the list and inserting n at the position $List.count - i$

We can see that our case is slightly more complicated as the example with taking steps as we need to calculate the position for the *List.insert* function anew for both cases. However, the principle still applies that we are changing the order of the function calls and get to the same result. We will make use of the previous definition of C where each element is part of a set Y. Further, we introduce i as an index and n as the element to insert, which is also part of the set Y. We can then write a property as follows:

$$\begin{aligned}\forall C, \forall n, i = 0, .., List.count(C) :& List.reverse(List.insert(C, i, n)) = \\ & List.insert(List.reverse(C), List.count(C) - i, n)\end{aligned}$$

The property is also shown in Listing 2.3 with example values. In the first case, we add four after the first position into the list $[1, 3, 7, 6]$ and receive $[1, 4, 3, 7, 6]$ and then reverse it to receive $[6, 7, 3, 4, 1]$. In the second case, we first calculate our index to make things easier to explain. We subtract the position (one) from the number of elements in our original list (four) and receive three as the index to insert. We reverse the list to $[6, 7, 3, 1]$ and insert the element four after the third element to receive $[6, 7, 3, 4, 1]$. In both cases, we receive the same final result. Again as for the last property, we test this one with an appropriate amount of random inputs and assume that the property holds.

Another important feature of QuickCheck is shrinking. Randomly generated test inputs usually contain some amount of irrelevant data (noise) which have no influence on the fact that the test case is failing. QuickCheck tries to reduce this noise through shrinking the input values and creating a more comprehensive and simpler counterexample [71].

There exists a wide range of QuickCheck adaptions for other languages. Some of those tools support stateful testing and therefore enable MBT. In the work of Hughes [52] test cases

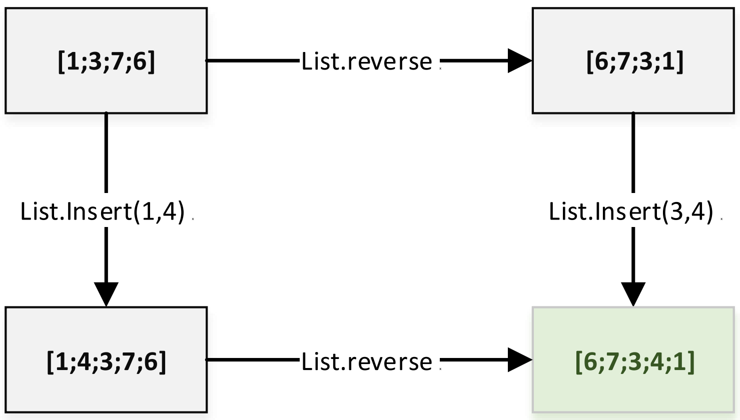

Figure 2.3: Combining functions in a different order might lead to the same final result.

consist of a sequence of operations which can execute functions with side-effects. After a test is run, it should be made sure that the test finishes in a clean state in order to start the next sequence of operations from a consistent initial state.

2.2.1 QuickCheck Inspired Tools

Some important tools to note are Quviq QuickCheck [16], a re-design for Erlang, ScalaCheck [83], a library written in Scala, SmartCheck [85], a Haskell library that builds on the QuickCheck back-end and generalizes shrunk values and FsCheck [96], a tool written in F# for .NET. FsCheck will be used in this thesis and is explained in more detail in Chapter 3.

2.3 Business Rule Engines

This section describes the term business rule and business process, how they differ and presents languages and tools to help model and embed them in software.

2.3.1 Business Rule and Business Process

A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business [101]. An important characteristic of business rules is that they tend to change often, more often than the underlying business object [15]. The following sentence is an example of a business rule.

A customer gets a 10 percent discount if he orders more than five times. Embedding business rules into the application process is still common practice. This has a negative impact on the maintainability of business logic because business rules tend to change quite frequently. Separating the business rules from the application code greatly improves the maintainability [92].

A business process clearly describes the work performed by all resources involved in creating outcomes of value for its customers and other stakeholders [26].

Ross [93] states that a rule is not a process and not a procedure. Rules should not be contained in either of these. Rules apply across processes and procedures. Business-rule management-systems generate assertions or take actions based on business rules [48]. Business-process management-systems take an application through stages in a workflow [95].

It is important to note that workflow-based applications sometimes incorporate embedded rule engines. Business rule engines can also embed workflow details.

2.3.2 Tools

There exists a vast variety of tools that embed processes and rules in business logic. The most notable languages are the Business Process Modeling Language [14] and the Business Process Execution Language [13], which is based on IBM's language: Web Service Flow Language [70] and Microsoft's language: Xlang [100]. Often those languages are used to enable coupling of web-services from different providers as shown in the studies of Rosenberg and Dustdar [91] and Charfi and Mezini [31].

In our case study business rules are stored in XML files as BRMs. The abstract models needed for MBT are derived from these models. The BRMs in the case study are a custom implementation and will be explained in Chapter 5.

2.4 Coverage Criteria

Code coverage is a measure that describes to which degree the source code of a program is executed (covered). The metric is widespread in any stage of testing. Test suites that never execute a certain code fragment will most likely not reveal any faults lurking there. Code coverage can be measured in a variety of ways. This includes structure-based coverage such as statements, lines, conditions, methods and classes [112]. This can also include more complex coverage criteria such as not only considering if a condition evaluates to true or false but also the evaluation of all the sub-expressions of the condition. Dataflow-based criteria can evaluate if a test case covers a certain $def - use$ association of a variable. This is a pair consisting of a definition and a use-location, such that there is a path between them that does not redefine or undefine the location [56].

Code coverage of a test suite can indicate its fault detection capability. A test suite with high code coverage has more of its code executed which suggests a lower chance of containing undiscovered faults. The study of Inozemtseva and Holmes [57] presents that a high correlation between coverage criteria and effectiveness exits when the size of the test suite is not controlled. However, when size is controlled the correlation drops. For the controlled approach test suites were compared maintaining the same constant test coverage while measuring other metrics. In the uncontrolled approach, the coverage was not applied to limit the test suites. Their results suggest that coverage should not be used as a quality target. Different coverage criteria such as branch and state coverage have shown similar results. Code coverage-criteria can only be applied in white-box testing-approaches, which presents another significant limitation. But most importantly, coverage criteria can only measure existing code. They are not suited for indicating faults arising from missing code.

2.4.1 Mutation-Based Criteria

A widespread measurement for test suite effectiveness is the mutation score as described previously in Section 2.1. In the empirical studies of Andrews et al. [12] and Just et al. [63] it was found that the coupling effect between real faults and mutants exists in general, even when code coverage is controlled. They also state that some faults are not covered by typical mutation operations such as faults that are caused by extra code or algorithms that do not work correctly. Non-coupled faults amount for 17 percent of the mutants. This shows the limitation of current mutation analysis. Even though mutation-based approaches show limitations, mutation score should be a better predictor of real fault detection than code coverage.

2.4.2 Model-Based Criteria

Since this research presents an MBT approach for generating a test suite, it is important to consider test selection criteria for models. Utting and Legeard [107] describe the major test selection criteria that are common in MBT, how to combine them and present case studies. One group of prominent criteria are the transition-based coverage-criteria. They are structural criteria that have been developed for transition-based modeling notations such as finite state machines, extended finite state machines and LTSs.

Transition-based models are made up of states and transitions. Depending on the notation a transition may be labeled with various kind of information. The most common criteria are the all-states coverage and the all-transitions coverage. Every state/transition of the model has to be visited at least once.

In recent studies, model-based and specification-based criteria were used and showed good results [10, 79, 94]. However, no exhaustive surveys on how MBT criteria perform compared to others were found.

3 Testing with FsCheck

In this section, FsCheck is explained by showing examples of properties. Writing properties can be seen as writing a testable specification of the program. FsCheck is written in F#, however, examples provided in this section will be written in C# unless otherwise stated. The explanation and examples are based on the FsCheck version 2.5.0. Since the tool is still under development future functionality might differ from the one described here.

3.1 A Few Simple Examples

In our examples, we are making use of lambda(λ) expressions. A λ-expression is an anonymous function. It can be written locally and passed as an argument. The theoretical background of those expressions is λ-calculus. It is a formal system based on function abstraction and application. For example, the following λ-term $\lambda. * 2x$ is a λ-abstraction for the function $f(x) = x * 2$. It is an anonymous function taking a single input x and substituting it with the term $x * 2$ [18]. One can see the similarity to the following C# syntax `Func<int,int> times2 = x => 2 * x`. One main difference is that in C# we have to define the type for the variables used. Both input and output types have to be defined. A more detailed explanation of λ-expressions for C# can be found in Hilyard and Teilhet [50].

Example 1: We will consider the simple example presented in Section 2.2. We will test the function *Reverse* with a property that checks that the reverse of the reverse of a list is the list itself. This is shown in Listing 3.1. The listing consists of two parts. The first one is a function that represents our property. The second part is a for-all quantification over integer arrays that tests our property. The example can be read as follows: Test if for all integer arrays the property *revRevIsOrig* holds. FsCheck randomly generates integer arrays and checks if the property returns true for all cases. If FsCheck does not encounter an error we cannot assume that a property truly holds. It just means that FsCheck could not find an error for a number of randomly generated values. FsCheck may report false negatives. However, we can be fairly confident that a property holds if the generated values are meaningful and the number of tests is big enough. A method that checks a property with randomly generated input is referred to as a parametrized or generative test.

It is currently not possible to generate elements for generic types with FsCheck. In our previous example, we explained the *Reverse* function. It takes an `IEnumerable<TSource>` as input and returns the same type. Our property projects from integer arrays to boolean values. When writing FsCheck test-cases we have to limit ourselves to one type. We chose to implement the property for the type integer arrays. Therefore, we are not testing the full range of values the function *Reverse* supports. The user has to carefully choose the data types manually to test the function's behavior. If we change the input type from using integer arrays to float arrays, we will see that in C# the example behaves the same. However, if we take a small excursion into the F# language we will see that the example can behave differently. Since F# is also part of the .NET framework a lot of functions behave similarly to C# examples. However,

```
Func<int[],bool> revRevIsOrig =
        xs => xs.Reverse().Reverse().SequenceEqual( xs );
Prop.ForAll(revRevIsOrig).QuickCheck();
```

Listing 3.1: Check property: reverse of the reverse of a sequence is the original sequence.

```
let revRevIsOrigFloat (xs:list<float>) =
  List.rev(List.rev xs) = xs
Check.Quick revRevIsOrigFloat
```

Listing 3.2: List reverse property for floats.

```
public static int Pow(int x, uint n) {
  int y = 1; //result initialization
  while (true) {
    if ((n & 1) != 0) y *= x; // if n is odd multiply by x
    n = n >> 1; // position of the next n bit
    if (n == 0) return y; // if there are no more bits return y
    x *= x;  // exponent for the next n bit
  }
}
```

Listing 3.3: Calculation of x^n with binary decomposition for unsigned exponents.

the F# implementation in Listing 3.2 FsCheck can falsify our property. Floating-point numbers can have the value NaN (not a number) and since $NaN \neq NaN$ comparing them with the equality operator will falsify the property with the simple array $[NaN]$. We could also use the *SequenceEqual* function as in C#, but since F# has by default implemented structural equality operators, we usually use those. In C# we usually do not use the equality operator for collections since they usually compare the references of the collections. Therefore, we used *SequenceEqual* in our C# example, which calls the *Equal* method for each element, which further returns true if both elements are NaN [96] [1]. This example should give us an insight as to why it is important and also very challenging to find data types to test our properties.

Finding meaningful properties can be a challenging task. For testing a function that adds two numbers using the + operator it is not advisable to apply the same operator in the test case. Testing an implementation with duplicating the code is usually a bad idea. In our previous example, we created a property based on the function's inverse. This is a useful pattern for several operations such as serialization/de-serialization read/write. An explanation of patterns for finding properties can be found in Wlaschin [110]. Halleux and Tillmann [47] present common test patterns for Microsoft's Pex [103]. Some of these patterns explained are applicable for PBT as well.

Example 1: In the next example we will test an exponentiation function. We will analyze the implementation of [62] as shown in Listing 3.3.

This time, to find our properties, we will take a look at mathematical definitions using elementary algebra. Mathematical properties are often a good basis to implement properties for FsCheck and can often be easily found in the literature. The four properties that will specify exponentiation are the following:

1. Initial condition: $b^1 = b$
2. Recurrence relation: $b^{n+1} = b^n * b$
3. Associativity $b^{n+m} = b^n * b^m$
4. Zero exponent $b^0 = 1$

A parameterized test that checks the four properties is shown in Listing 3.4. The method

[1]https://referencesource.microsoft.com/#mscorlib

```
[TestMethod]
public void PowTestFsCheck() {
  Func<int, bool> ZeroExponent = b => Pow(b, 0) == 1;
  Func<int, bool> InitialCondition = b => Pow(b, 1) == b;
  Func<int, uint, bool> RecurrenceRelation = (b, n) =>
    Pow(b, n + 1) == Pow(b, n) * b;
  Func<int, uint, uint, bool> Associativity = (b, n, m) =>
    Pow(b, m + n) == Pow(b, n) * Pow(b, m);

  Prop.ForAll(ZeroExponent).QuickCheckThrowOnFailure();
  Prop.ForAll(InitialCondition).QuickCheckThrowOnFailure();
  Prop.ForAll(RecurrenceRelation).QuickCheckThrowOnFailure();
  Prop.ForAll(Associativity).QuickCheckThrowOnFailure();
}
```

Listing 3.4: Test of properties for exponentiation using FsCheck.

```
[TestMethod]
public void MultPropertyTest() {
  Configuration config = Configuration.VerboseThrowOnFailure;
  config.Replay = FsCheck.Random.mkStdGen(50);
  Func<int, int, bool> WrongMultiplyProperty = (a,b) =>
    a == 0 || b == 0 || a * b >= a;
  Prop.ForAll(WrongMultiplyProperty).Check(config);
}
```

Listing 3.5: Wrong multiplication property which shows how FsCheck shrinks values.

is annotated with *TestMethod*. A unit-test framework is, therefore, able to identify the method as a test case. FsCheck will throw an exception if one of the properties does not hold. Each property is run with 100 randomly generated inputs by default. The first tests start with small values and get larger the closer they are to the last test case. Properties can also be written in a class as a static method. FsCheck supports checking all public methods on the given type that have a testable return type.

3.2 Shrinking

Example 1: We will now look at what FsCheck does if it encounters an error. A property for multiplication is shown in Listing 3.5. Using the Configuration *VerboseThrowOnFailure* has two implications. If it finds a counterexample a *System.Exception* is thrown and every test case up to this point including the failing one is printed. Afterwards, execution is stopped. The *Quick* Configuration would only print the failing test case. Note that we set the seed of the random generator in order to receive the same test cases every time we run the test. This can be helpful for debugging.

We defined the property *WrongMultiplyProperty* wrong as we were only considering positive integers. However, we defined it for all integers as shown in the following equation:

$$\forall a, b \in \mathbb{Z} : a \neq 0 \land b \neq 0 \rightarrow a * b \geq a$$

```
Falsifiable, after 3 tests (2 shrinks) (StdGen (51,1)):
Original:
(-2, 3)
Shrunk:
(-1, 2)
```

Listing 3.6: Exception message of wrong multiplication property.

```
public class MyArbs {
  public static Arbitrary<string> StringLowerCase() {
    return Arb.From(Arb.Default.String().Generator
      .Where(s => s != null && Regex.IsMatch(s, "^[A-Z]+\$")));
  }
}
```

Listing 3.7: An upper-case string generator using the *Where*-filter.

Cases where a is negative and b is bigger than 1 will produce a negative number which is always smaller than a. An example of an exception message that FsCheck might throw is shown in Listing 3.6.

Original is the counterexample FsCheck has found. *Shrunk* shows FsCheck's attempt to find the minimum counterexample that still fails the property. In our case it succeeded, `(-1,2)` is indeed minimal. A shrinker for the data type T has the signature `Func<T,IEnumerable<T>`. Given a value, a shrinker produces a sequence of smaller values. If FsCheck finds a value in the sequence that still produces the same error it will take this value as the new counterexample and continue its recursion until it cannot find any smaller values. This local minimum is then the value *Shrunk* in the exception message.

3.3 Generating Test Data

The test data FsCheck provides is produced by test data generators. FsCheck defines default generators and shrinkers for common types. Both are packed together in an *arbitrary* instance. FsCheck reads the method signature of the property to test and performs a lookup on the parameter types to identify which *Arbitrary* instance it has to select. FsCheck is also able to generate test data for discriminated unions, record types, and enumerations by default. However, if the user wants to use a custom arbitrary the *Arbitrary* needs to be explicitly added to FsCheck in order to be found during a lookup on the parameter types. This principle is called registering and is done via the command `Arb.Register<T>();`, where T is the type of a class that defines the arbitraries as static members. Registering is usually needed for custom types since they neither have a default generator nor shrinker.

3.3.1 Custom Generators

Example 1: In the next example different ways to generate only upper-case strings are shown and compared.

The first approach filters values after they are generated. Later in this section, it will be shown why this is not a good solution for this specific problem. An example implementation using regular expressions is shown in Listing 3.7.

```
1  public static Gen<string> UpperCaseStringGenerator(int size) {
2    Gen<int> intGen = Gen.Choose('A', 'Z');
3    var gens = new List<Gen<char>>();
4    for (int i = 0; i < size + 1; i++) {
5      gens.Add(intGen.Select(c => (char)c));
6    }
7    Gen<IEnumerable<char>> charSeqGen = Gen.Sequence(gens);
8    return charSeqGen.Select(cs => new String(cs.ToArray()));
9    });
10 }
11 public static Arbitrary<string> ArbUpperCaseString() {
12   return Arb.From(Gen.Sized(i => UpperCaseStringGenerator(i)));
13 }
```

Listing 3.8: String generator for upper-case strings using Gen.choose.

```
[TestMethod]
public void UpperCaseTest() {
  Arb.Register<MyArbs>();
  Func<string, bool> AllUpperCaseLetters = s => Regex.IsMatch(s, "^[A-Z]+\$");
  Prop.ForAll(AllUpperCaseLetters).VerboseCheckThrowOnFailure();
}
```

Listing 3.9: Test case that shows the use of a registered arbitrary.

The second approach does not take the default string generator but rather defines its generator using `Gen<int> Choose(int l, int h)`. Generators are usually built from this function. It makes a random choice between the interval of l and h inclusive. The generator and arbitrary are shown in Listing 3.8.

As stated previously if a property is checked it starts with small examples and increases its size over time. The standard range is between 0 and 100. Therefore, we use $size + 1$ in the for-loop to ensure no strings with length 0 will be created. The lower and upper bound can be changed in the *Configuration* with which the property is executed. This size parameter is passed to our generator as shown in Line 12. Our generator creates a sequence of char generators which can only produce values between the chars 'A' and 'Z'. We then sequence the given list of generators into a generator of a list in Line 7. With *Select*, we then map from the generator of a char sequence to a string generator.

A test case using our custom string generator is shown in Listing 3.9. First, we register one of our previously defined custom arbitraries. Then we create a function which will serve as our property and finally we check if the property holds for all strings.

We will see two main differences between a test case using a *Where*-filter and one using a custom generator if we look at the output of both test cases. The first difference is the length of the generated strings. The second one is a speed difference. Since we limited the space of possible strings significantly the first approach has hard times finding appropriate strings. It simply tries to generate a string and if it fails it tries again. Therefore, the length of the strings is small and the computation is expensive. It produces almost no string that has a length bigger than 3. The speed difference for 100, 1000 and 10000 generated tests is shown in Table 3.1. Filters should only be applied if the purpose is to filter out a few examples, like the following: $length \neq 0$ or $s \neq null$.

Number of tests [#]	*Where*-filter [ms]	Custom generator [ms]
100	854	127
1000	6 968	159
10000	68 511	477

Table 3.1: Computation time to test a property with upper-case string generators.

```
return Arb.From(Arb.Default.String().Generator
  .Where(s => s != null && Regex.IsMatch(s, "^[A-Z]+\$"))
  , Arb.Default.String().Shrinker);
```

Listing 3.10: Extension of upper-case string generator by standard shrinker.

One problem remains with both approaches. What if a test case needs a default string and a string that consists only of upper-case letters. Since we replaced the default arbitrary for strings with a custom one by registering our custom arbitrary FsCheck can no longer generate "default" strings. The solution is to define a struct in C# or a record type in F# that contains a string and define our arbitrary for the custom type. With `Arb.Register<UpperCaseString>();` we register a custom struct which we can then use in our properties. The struct is shown in the following line:

```
public struct UpperCaseString { public string s; }
```

3.3.2 Custom Shrinking

We continue the current example to show how shrinking works for custom generators. Since we overwrote the standard string arbitrary with a generator only, shrinking is disabled. We try to register our custom arbitrary with the standard string shrinker as shown in Listing 3.10.

If we test the property

```
Func<string, bool> StringAreMaxLength2 = s => s.Length < 2;
```

we will receive a counterexample like *"JTA"* which is then shrunk to *"aa"*. FsCheck obviously treats generator and shrinker separately. Since we do not want to have shrunk values that are out of range of our generator we have to ensure ourselves generator and shrinker map to the same value space. An example shrinker for *UpperCaseString* is shown in Listing 3.11.

The first for loop removes one element of the string at each possible position and yield returns them. Yield return returns each element, one at a time. It is not necessary to define an *IEnumerable* instance. The second loop decreases the char value at each position by one with 'A' being the lower bound and yield returns them. Our custom shrinker works in two dimensions where one is the length of the strings and the second one is the value of the char elements, both using correct lower bounds to ensure the same value space as the generator. This time for the counterexample *"JTA"* we will receive the shrunk value *"AA"*, which is minimal for our upper-case string type.

3.4 Model-Based Testing

This section describes FsCheck's model-based testing functionality. Currently, two approaches for Model-based testing (MBT) are available in FsCheck. The first one can be seen as the regular version, which was implemented first. The second one is still in an experimental

```csharp
public static IEnumerable<UpperCaseString> Shrinker(UpperCaseString ucs) {
  string s = ucs.s;
  for (int i = 0; i < s.Length; i++) {
    yield return new UpperCaseString(s.Remove(i, 1));
  }
  for (int i = 1; i < s.Length; i++) {
    if (s[i] > 'A') {
      var ca = s.ToCharArray();
      ca[i] -= (char)1;
      yield return new UpperCaseString(new string(ca));
    }
  }
}
```

Listing 3.11: Custom shrinker of upper-case string generator

namespace since it is not completed yet. This thesis mainly makes use of the second version and therefore the explanation focuses on the experimental version.

3.4.1 Procedure

In order to perform MBT with FsCheck, certain components are needed. A system that will be verified, called system under test (SUT). A model, which is an abstraction of the SUT and describes a partial behavior of the SUT. Several operations which will be executed on the SUT and on the model and a test framework which handles the execution of operations, or command as it is called in some other PBT tools, and verifies pre- and post-conditions. The test framework is already supplied by FsCheck. The user has to supply the other parts and configure the test framework.

Figure 3.1 shows the flow-chart of one test execution of FsCheck. First, a *Setup* is performed which sets the initial state of the model and the SUT. Since the initial state is generated, we can set a randomized initial state. For example, the initial amount on a bank account can be set to a value in the range of the bank accounts limits.

The test framework evaluates if the maximum number of operations is already reached. The number of maximum operations can be configured before the test-case execution. If the maximum number is not reached, FsCheck will try to generate a new operation. The specific operations are selected in *Next*. The function *Next* has a generator and the current model as input and returns a set of possible operations. *Setup* and *Next* are both located in an own class called *Machine*.

FsCheck tries to find an operation that satisfies its precondition. If no precondition can be satisfied after a given number of tries the test case is aborted. This check is performed in the operation's function *Pre* which receives the current model and decides if it is possible to execute the current operation. For example, it might be impossible to withdraw money from a bank account that is empty. If an operation is found that satisfies its precondition the operation is run. The *Run* function of the operation receives the current model, modifies it and returns a new model. The three function of *Next*, *Pre*, and *Run* are performed in a loop until the maximum number of operations is reached.

After finishing the loop the previously run operation has to be checked. One by one each operations' *Check* function gets called. In *Check*, the SUT is exercised. The model, which was returned from *Run*, is available in this function. The property (postcondition) of the operation

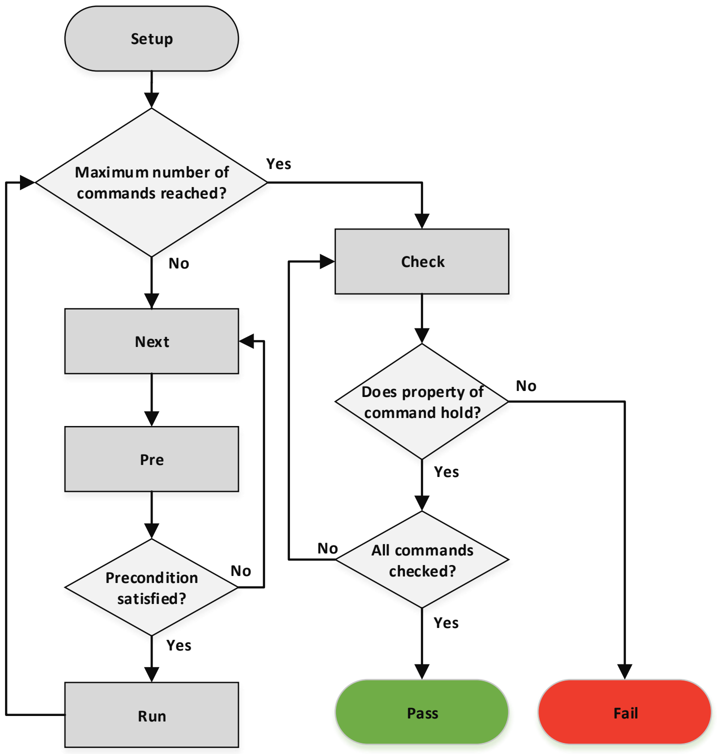

Figure 3.1: Flowchart of a test case executed by FsCheck's state machine module.

```
public class BankAccount {
  public int Money { get; private set; }
  public BankAccount(int initialMoney) { Money = initialMoney; }
  public void Deposit(int inc) { Money += inc; }
  public void Withdraw(int dec) { Money -= dec; }
  public override string ToString()
    { return String.Format("BankAccount: {0}", Money); }
}
```

Listing 3.12: Bank account as a system under test.

is verified here. This is usually done by comparing the state of the model with the state of the SUT. If a property does not hold, the verdict of the test case is fail. If the properties of all operations hold the verdict is pass.

3.4.2 Example: Bank Account

The example of a bank account will illustrate how MBT can be performed with FsCheck. Listing 3.12 shows the SUT of this example. In a real-world example, we have to imagine complex operations behind the logic of a bank account. However, the example shown here attempts to be minimal. A bank account gets initialized with a money value and two methods can be executed on it. *Deposit* to put money on the account and *Withdraw* to take away money from the account. FsCheck will call the *ToString* method to log the state of the object during a test run.

FsCheck defined an interface, `Machine<SystemUnderTest,Model>`, containing method signatures for *Setup* and *Next*. We will abstract our *BankAccount* as an integer number representing

```
public class BankAccountMachine : Machine<BankAccount, int> {
  public override Arbitrary<Setup<BankAccount, int>> Setup
    { get { return new BankAccountSetupArb(); } }
  public override Gen<Operation<BankAccount, int>> Next(int value)
    { return ArbsToRegister.BankAccountOperationArb().Generator; }
}
```

Listing 3.13: Bank account machine

```
public class BankAccountSetup : Setup<BankAccount, int> {
  public int Initial { get; }
  public BankAccountSetup(int inital) { Initial = inital; }
  public override BankAccount Actual() { return new BankAccount(Initial); }
  public override int Model() { return Initial; }
}
public class BankAccountSetupArb : Arbitrary<Setup<BankAccount, int>> {
  public override Gen<Setup<BankAccount, int>> Generator {
    get {
      return Gen.Choose(0, 100).Select(i
        => (Setup<BankAccount, int>)new BankAccountSetup(i));
    }
  }
  public override IEnumerable<Setup<BankAccount, int>>
  Shrinker(Setup<BankAccount, int> _arg1) {
    var initials = Arb.Default.Int32().Shrinker(((BankAccountSetup)_arg1).Initial);
    foreach (var i in initials) {
      yield return new BankAccountSetup(i);
    }
  }
}
```

Listing 3.14: Bank account setup and arbitrary

the money on the bank account. The `Machine<Bankaccount,int>` is shown in Listing 3.13.

The function *Setup* returns an *Arbitrary* of the type *Setup*. A Setup is a class that contains two functions. One that returns the initial state of the SUT called *Actual* and another one that returns the initial state of the model called *Model*. The Setup, as well as the arbitrary, are shown in Listing 3.14. Arbitraries were explained previously in Section 3.3, here we focus on the functionality of MBT. Note that shrinking is supported through the property-based nature of the tool.

The function *Next* has a generator as input to generate an operation that will be performed on the SUT and the model. It is useful to put this generator into an arbitrary, implement the *Shrinker* and register the arbitrary. This allows FsCheck to not only shrink the operation sequence in a test case but also shrink the operations themselves. In our case, a *WithdrawOperation* that withdraws an amount of 50 from the bank account could be shrunk to one that withdraws only one. The generator in our example chooses between a *WithdrawOperation* and a *DepositOperation* which can have a transaction amount between one and ten, inclusive. This is shown in Listing 3.15.

The last parts we need are the operations. In Listing 3.16 the *DepositOperation* is shown.

```
public override Gen<Operation<BankAccount, int>> Generator
{
  get {
    var incGen = Gen.Choose(1, 10).Select(i
      => (Operation<BankAccount, int>)new DepositOperation(i));
    var decGen = Gen.Choose(1, 10).Select(i
      => (Operation<BankAccount, int>)new WithdrawOperation(i));
    return Gen.OneOf(incGen, decGen);
  }
}
```

Listing 3.15: Generator function of the bank account operation arbitrary.

```
public class DepositOperation : Operation<BankAccount, int> {
  public int Amount { get; private set; }
  public DepositOperation(int amount) { Amount = amount; }
  public override bool Pre(int m) {
    if (m + Amount > 100)
      return false;
    return true;
  }
  public override int Run(int m) { return m + Inc; }
  public override Property Check(BankAccount a, int m) {
    a.Deposit(Amount);
    return (a.Money == m).ToProperty();
  }
  public override string ToString() {
    return String.Format("Deposit:{0}", Deposit);
  }
}
```

Listing 3.16: Implementation of the deposit operation.

We want our bank account to remain between an amount of 0 and 100. We ensure that the bank account is not tested outside these boundaries via preconditions. The *Run* function simply adds the amount that we want to deposit to our model. *Check* exercises the SUT and compares the model with the value of the bank account. This represents our postcondition and is returned as a property. The *WithdrawOperation* is implemented in a similar manner.

In order to execute our specification, we transform our machine into a property with the provided FsCheck functionality.

```
new BankAccountMachine().ToProperty().Check(Configuration.VerboseThrowOnFailure);
```

By converting the machine into a property common configurations can be applied. It is important to note that configuring the size of the generators will also influence the length of the test cases, e.g. setting the *EndSize* to 10 implies that no test case will have more than 10 operations. If we now execute the above line in a test method we will get output for each test case. Test case outputs will have the form as shown in Listing 3.17.

The first line shows the test case number. This one is the 10th test case out of 100 since the indexing is zero-based. In the second line, we see the initial state of our model, which is 92.

The following lines contain the output of the *ToString* format of the operation and the

```
9:
(92, Setup BankAccount)
Deposit:6 -> 98
Withdraw:2 -> 96
Withdraw:3 -> 93
Withdraw:1 -> 92
Withdraw:5 -> 87
Deposit:2 -> 89
Deposit:1 -> 90
Deposit:8 -> 98
Withdraw:10 -> 88
Deposit:3 -> 91
Withdraw:10 -> 81
```

Listing 3.17: Test case output of the bank account specification.

model value after execution of the operation. In this example, the first operation deposits an amount of six. With the initial amount of 92, the bank account should hold an amount of 98 after execution of the operation. Note how defined limits are adhered to. Almost no *Deposit* operations were executed in the beginning of our test case since the balance is close to the upper limit of 100.

As a next step, we show how FsCheck deals with a faulty implementation. A simple artificial bug is implemented in the *Withdraw* functionality of the SUT. Withdraw only works if the amount on the bank account is lower than 50. This is realized by adding the following line `if (Money >= 50) return;` to the function *BankAccount.Withdraw*.

Upon testing the faulty implementation a counterexample for the specification was found after 10 test runs. Table 3.2 shows the original found counterexample as well as the shrunk counterexample. The identifiers of the operations were tracked in order to show how the sequence of operations was shrunk and which operations were shrunk.

Shrinking of operations themselves is not possible by default. This is only enabled if an arbitrary for the operation is implemented and registered at FsCheck. The last operation of the counterexample was shrunk to its minimum. With the test output, we are able to make an educated guess about the error. The error occurs on a *Withdraw* operation if the amount of the bank account is higher or equal to 50 and is not influenced by the amount the operation withdraws.

Original	Shrunk
Deposit:5 $\rightarrow$ 33	
Deposit:6 $\rightarrow$ 39	Deposit:6 $\rightarrow$ 34
Deposit:2 $\rightarrow$ 41	Deposit:2 $\rightarrow$ 36
Deposit:1 $\rightarrow$ 42	Deposit:1 $\rightarrow$ 37
Deposit:2 $\rightarrow$ 44	Deposit:2 $\rightarrow$ 39
Deposit:10 $\rightarrow$ 54	Deposit:10 $\rightarrow$ 49
Deposit:1 $\rightarrow$ 55	
Withdraw:1 $\rightarrow$ 54	Deposit:1 $\rightarrow$ 50
Withdraw:3 $\rightarrow$ 51	
Withdraw:3 $\rightarrow$ 48	Withdraw:1 $\rightarrow$ 49
Deposit:3 $\rightarrow$ 51	

Table 3.2: Original and shrunk counterexample of a faulty bank account implementation.

With meaningful string representations and shrinking mechanism for operations, it is often possible to narrow down the type of error in a meaningful way and to reduce debugging time. This also holds true for more complex examples. However, during the thesis, it was observed that shrinking is not always feasible in real-world examples. It is not always possible to shrink an example even if the error could be found with a shorter sequence. In order to shorten some sequences, multiple shrink operations would have to occur in one step.

Running long sequences on a real system is usually slower due to the communication of web-services, database accesses, control of hardware components and other complex and time-consuming tasks. Since shrinking tries to remove one operation per shrinking step from the sequence this might result in long run times. Shrinking is a very useful tool to narrow down errors but it is important to keep in mind that real-world systems may operate to slow to make shrinking feasible and that a minimal counterexample will not always be found.

For further information on how MBT with FsCheck works and other examples refer to the user manual of FsCheck[2]. The examples in the manual are written mainly in F#. While some C# examples exist, they do not for the experimental MBT version.

It is however interesting to compare the style between the F# and C# version. The F# version benefits from a powerful type inference system, which makes the code snippets concise. This can be observed in generic type parameters, which have to be written explicitly in the C# version and in the need of defining classes which contain a huge amount of boilerplate code as seen in the *Setup* class.

[2]`https://fscheck.github.io/FsCheck`

4 Testing with MoMuT

MoMuT (MOdel-based MUtation Testing) is based on mutation testing. Instead of programs, abstract models of the SUT are mutated. A model can be represented as an UML state-transition diagram or as an object-oriented action system (OOAS). Then faults are introduced into the original model via mutation operators. The objective of MoMuT is to identify and kill these mutated models.

This Chapter will give an overview of the architecture of MoMuT, explain OOAS and abstract test-case generation in more detail. The approach of this thesis interacts with MoMuT by providing the OOAS as input and interpreting the output of MoMuT in the form of abstract test-cases.

4.1 Architecture

MoMuT::UML is a mutation testing tool which takes UML state-transition diagrams as input. They are transformed into action systems and later interpreted as labeled transition systems (LTS)s. The test-case generator then performs a conformance check. The tool explores the state space of the original and mutated model and tries to detect differences in their behavior. If non-conformance is detected, a test case is generated that shows the difference. Model-based mutation testing (MBMT) is a computationally expensive strategy. Numerous mutants have to be analyzed and generating test cases involves a conformance check of the two models [3].

Figure 4.1 [67] shows the architecture of the tool. The input for MoMuT::UML are UML models. They can be modeled using tools such as Papyrus MDT[1] or Visual Paradigm[2]. The UML model is then converted to an object-oriented action system (OOAS). OOAS are explained in Section 4.2. The tool also mutates the UML model and converts these mutants to OOAS as well. It is possible to use only certain components of MoMuT. For example, OOAS models can be directly translated with the OOAS compiler without the need to define the models in UML. However, if this method is chosen mutation operator have to be implemented by the user since the UML mutator would not be applied to the models. In this thesis, we implemented our own simple mutation operator. This is further explained in Section 6.3.

For the test-case generation, three different back-ends are available. The ”enumerative back-end”, developed in the MOGENTES[3] project, the ”symbolic back-end”, developed within the TRUFAL[4] project and the 3rd generation version developed by the Austrian Institute of

[1]`https://eclipse.org/papyrus`
[2]`https://www.visual-paradigm.com`
[3]`http://www.mogentes.eu`
[4]`https://trufal.wordpress.com`

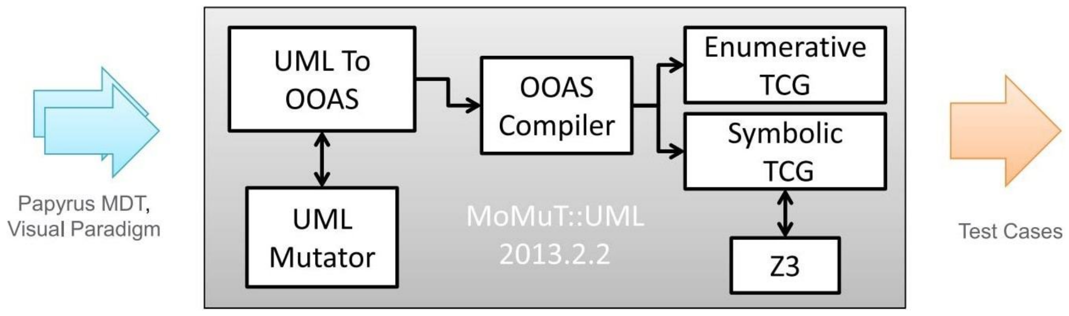

Figure 4.1: MoMuT::UML architecture overview.

Technology (AIT)[5]. The latest version uses UML models as input for the front-end, converts them to OOAS models for the back-end and uses the OOAS models for mutations and abstract test-case generation [82]. At the beginning of the thesis, the back-end was not available standalone. This would have required generating UML instead of OOAS models, which was out of the scope of the thesis. Between the enumerative and symbolic back-end, the latter showed better performance for the action systems in this thesis. Therefore, the symbolic back-end was chosen for the case study. For research beyond the scope of the thesis, we would recommend using the latest version of MoMuT as it seems to be the most promising.

All back-ends support the following three test-case generation strategies:

1. *Random*, which uses a random walk to traverse the original model and does not rely on mutants.
2. *Mutation*, which checks if the behavior of the mutant differs from the original and generates test cases that display this difference.
3. *Combined strategy*, which uses both, random and mutation.

The symbolic back-end uses Microsoft's SMT Solver Z3 [81]. It is written in SICStus Prolog [27] and is further explained in Section 4.3. Finally, MoMuT::UML produces abstract test cases in the Aldebaran format. The Aldebaran format is a file format for presenting LTSs [41].

In this thesis, a web-service is tested which defines its specification in business rule-models. These models are translated directly into extended finite state-machines and further into OOAS models and not into UML models. Therefore, the UML Mutator and the translation step from UML to OOAS of the tool are not used. Furthermore, the symbolic back-end was chosen as a test-case generator to generate abstract test cases. In the following chapters mainly the OOAS compiler and the symbolic back-end are explained. Therefore, MoMuT will refer to those components from MoMuT::UML.

4.2 Object-Oriented Action Systems

Action systems (AS) were first introduced by Back and Kurki-Suonio [17] as a modeling formalism for distributed systems. Action systems start at an initial state. The state is changed by executing actions on the system. One action is chosen at each step in a non-deterministic manner. Actions can only be chosen if they are enabled. They are enabled if their guard is satisfied in the current state. If there is no action enabled, the execution terminates. The Object-oriented action system (OOAS) language is an object-oriented extension which is based on the work of Bonsangue et al. [25].

4.2.1 Action System

There exist various different versions of the original AS notation by Back and Kurki-Suonio [17]. According to Jöbstl [61] the following notation to present an action system AS is common.

$$A = |[\textbf{\textit{var}} \quad v, w^* \bullet S_0, \textbf{\textit{do}} \quad A_1[]...[]A_n \quad \textbf{\textit{od}} \quad]| : I$$

An AS consists of the variable block var, an initialization action S_0 and the $do - od$ block. In the variable block, the set of variables v denotes internal variables of the AS. Variables marked with an asterisk w^* are exported by the action system and become global variables. Other AS can import them via the import list I. The central part of the action system is the do-od block. Repeatedly one of the contained actions $A_1, ..., A_n$ is chosen non-deterministically. An action

[5]https://www.ait.ac.at

A_i consists of a label definition, a parameter list, a guard and a body. The guard is a boolean expression which indicates if it is valid to execute the action in the current state. If the guard evaluates to true, the action is enabled. The body of an action is composed of statements that change the state of the variables in the AS. In addition to this common definition, actions will be marked to identify the type of action. The label definition may start with a question or an exclamation mark. A question mark denotes the action as input, an exclamation mark as output and the absence of both denotes it as an internal action [61, 99].

4.2.2 Object Orientation and Complex Data Types

An object-oriented action system consists of a finite set of classes, each class specifying the behavior of objects. They are dynamically created and executed in parallel [25]. OOAS also implements a prioritized composition operator, which has been introduced by Sekerinski and Sere [97], as in addition to the sequential and non-deterministic composition. Since these features are not required for the modeling process in this thesis they are not further explained. Complex data types, such as maps, lists, and tuples are also added to the OOAS language.

Listing 4.2 shows a part of the syntax of an OOAS. Certain parts of the language are not implemented in this thesis and therefore not further explained. For a more detailed syntax definition and explanation of the OOAS language, the interested reader may refer to Tiran [104].

In this thesis, an OOAS consists of a set of user-defined types and only one object of an AS class. In general, OOAS supports multiple instances of a class, however, since only one instance is needed in this work the original OOAS syntax is simplified to only reflect components relevant to this thesis. The AS has a set of variables that are initialized. Following is a set of actions that can be marked as input with *ctr*, as output with *obs* or internal actions if it is not marked. Each action consists of a guard and an assignment block. Finally, the do-od block contains the calls to the named actions.

There are several types of variables that are supported in the OOAS language in the type initialization block. Integer types must have a lower and upper bound. Types can also be defined by enumerating possible values and assigning them to integer types. The standard boolean type is also supported. Complex enumerable types are also supported as mentioned previously.

Listing 4.1 shows a bank account OOAS. A bank account model was as well introduced for FsCheck in the previous chapter 3. The example shown here contains, as before, the *Deposit* and *Withdraw* actions. They are marked as input actions. In addition, it also contains a *Notify* action which is marked as output. $Notify$ should inform the user that his account is almost depleted.

The order of type and variable definition is placed in different locations and the initialization is done in the variable definition block of a class. When a class is marked as *autocons*, one instance of the class will be created automatically. This object is called the root object. In our case, the AS consists only of one class and one object, the bank account.

The named actions of the class can be seen in the Lines 8 to 16. They can only be called from within the *do-od block* in Line 17 to 21 and may not be recursively nested. Each of the named actions has the form of a guarded command. This is seen in Lines 9, 12 and 15. The composition contained in the *do-od block* is non-deterministic and expressed with the following notation: $S_1[]S_2$. This means that any action can be chosen at each step as long as its guard is enabled.

$$\begin{aligned}
OOAS &:= 'types' \quad TypeList \quad 'system' \quad Identifier \\
TypeList &:= (Identifier \quad ' =' ComplexType \quad ';')* \\
& \quad Identifier \quad ' =' \quad OOActionSystem \\
OOActionSystem &:= 'autocons' \quad 'system' \quad Identifier \quad '|[' \\
& \quad 'var' \quad AttrList? \quad 'actions' \quad NamedActionList \\
& \quad 'do' \quad ActionCallBlock \quad 'od' \quad '|]' \\
AttrList &:= (Attr \quad ';') * Attr \\
Attr &:= Identifier \quad ' :' \quad ComplexType \quad ' =' Exp \\
ComplexType &:= SimpleType \quad | \quad EnumerableType \\
Exp &:= Atom \quad BinOperator \quad Atom \\
NamedActionList &:= (NamedAction \quad ';') * NamedAction \\
NamedAction &:= ('obs'|'ctr')?Identifier \\
& \quad '(' \quad MethodParamList \quad ')' \quad DiscreteActionBody \\
DiscreteActionBody &:= 'requires' \quad Exp \quad ' :' ActionBody \quad 'end' \\
ActionBody &:= ActionBodyParallel|ActionBodySeq|Statement \\
ActionBodyParallel &:= ActionBlockParen'[]' \quad ActionBlockParen \\
ActionBodySeq &:= ActionBlockParen';' \quad ActionBlockParen \\
ActionBlockParen &:= ActionBlock|DiscreteActionBody \\
Statement &:= ('skip') \quad | \quad (Reference \quad ' :=' Exp) \\
ActionCallBlock &:= (Action \quad '[]') * Action \\
Action &:= 'var' \quad Identifier \quad ' :' \quad ComplexType \\
& \quad ' :' \quad Identifier \quad '(' \quad MethodCallParam \quad ')' \\
MethodCallParam &:= (Exp \quad ',') * Exp
\end{aligned}$$

Figure 4.2: Definition of an object-oriented action system syntax used in this thesis.

4.3 Abstract Test-Case Generation

Abstract test-case generation is performed by MoMuT's symbolic back-end. The heart of the test-case generation process is conformance checking. It is checked if the mutated model conforms to the specification, i.e. the original model. If a state can be reached that does not conform to the specification the path to the failing state is saved and a test case is obtained [61].

4.3.1 Input-Output Conformance of Labeled Transition Systems

In order to perform the conformance check, the OOAS models are mapped to LTSs. Having an LTS interpretation of the models, it is possible to exploit established testing theories [2]. As noted previously actions are distinguished between controllable (input), observable (output) and internal actions (τ). Based on this LTS semantics an input-output conformance (ioco) check can be performed.

The ioco relation was first introduced by Tretmans [105]. The following notions are mainly based on this paper. An LTS is a 4-tuple (S, L, T, s_0) where

```
types
  PositiveInt = {0..100};
  BankAccount = autocons system
  |[
    var
     balance : PositiveInt = 50;
    actions
      ctr Deposit(inc : PositiveInt)
        requires balance + inc <= 100:
          balance := balance + inc;
      ctr Withdraw(dec : PositiveInt)
        requires balance - dec >= 0:
          balance := balance - dec;
      obs Notify(val : PositiveInt)
        requires balance < 10 && value == balance:
          skip;
    do
      var A : PositiveInt : Deposit(A) []
      var B : PositiveInt : Withdraw(B) []
      var C : PositiveInt : Notify(C)
    od
  ]|
system
  BankAccount
```

Listing 4.1: Bank account as an object-oriented action system

- S is a countable, non-empty set of states,
- L is a countable set of labels,
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation,
- $s_0 \in S$ is the initial state.

The labels L are the observable actions of a system and τ is a non-observable internal action. A transition $(s, \mu, s') \in T$ starts from the state s, has a label μ and ends in the state s'. It is denoted as $s \xrightarrow{\mu} s'$.

A **trace** σ is the sequence of observable actions of a computation, where a computation is a composition of transitions. The traces of a state $traces(s_n)$ denote all traces $\sigma \in L \cup \varepsilon$ starting from s_n, where ε is the empty sequence. It is common to refer to all traces of an LTS p starting from the initial state s_0 as $traces(p)$ instead of $traces(s_0)$. We further define $p \quad after \quad \sigma$ as the set of possible states after the trace σ from the start state p.

We can partition the labels L of an LTS into input L_I and output L_U actions where $(L = L_I \cup L_U) \wedge (L_I \cap L_U = \emptyset)$. We denote this new class of input-output transition systems as $IOTS(L_I, L_U) \subseteq LTS(L_I \cup L_U)$. Before explaining the ioco-relation a few other notions need to be introduced.

- If in every state, every input is enabled, then the automaton is said to be input-enabled, or **strongly input-enabled**. This means that the automaton is unable to block its input, which is an important assumption for the models [75].
- The notion from Tretmans ioco-relation also allows input enabling via internal transitions τ, which is called **weakly input-enabled** [105].

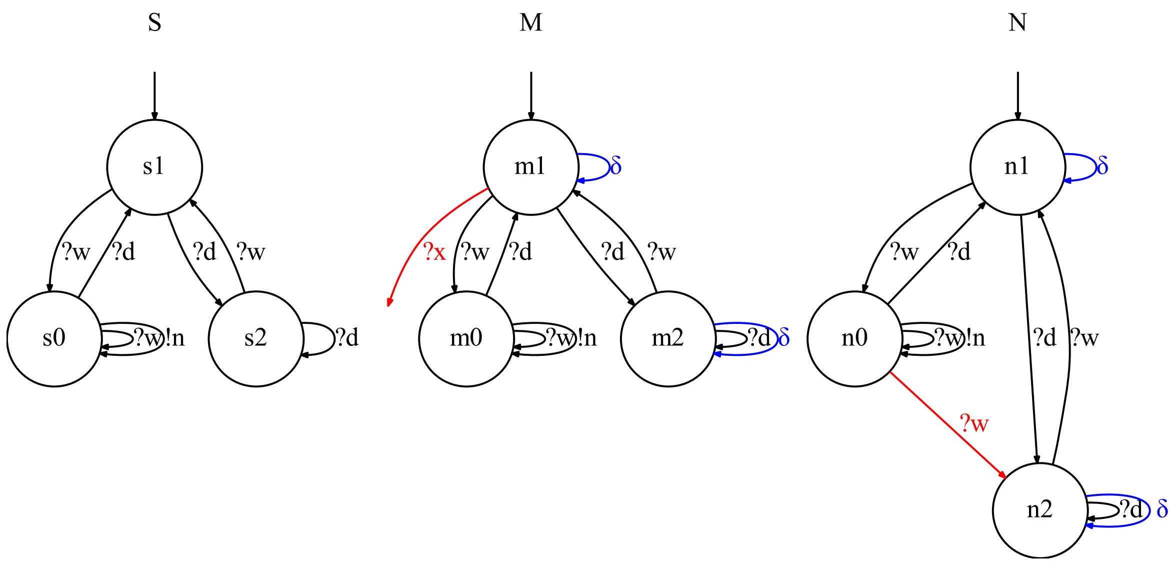

Figure 4.3: Examples for LTSs from mutated bank account models.

- If a state generates no output during a certain period of time, it is considered a **quiescent-state** and it contains a transition to itself labeled with δ [21].
- If an LTS eventually enters a quiescent state, it is called **strongly responsive**. This means that the system does not have any infinite paths consisting of only internal actions τ [21].

Let the mutated model $MM = (S^{MM}, L^{MM}, \rightarrow^{MM}, s_0^{MM})$ be a weakly input-enabled $IOTS$ and let the specification $S = (S^s, L^S, \rightarrow^S, s_0^S)$ be a strongly responsive LTS. Then, ioco can be defined as follows:

$$MM \quad ioco \quad S =_{df} \forall \sigma \in traces(S) : Out_{MM}(MM \quad after \quad \sigma_{MM}) \subseteq Out_S(S \quad after \quad \sigma_S)$$

After denotes the set of reachable states after a certain trace σ. **Out** denotes the set of all output events in a set of states S [5]. The mutated model is input-output conform to the specification iff the outputs of the mutated model are possible in the specification after the trace of the specification for each trace in the specification.

Example: The ioco check is shown on a simplified version of the bank account. As opposed to before the bank account is initialized with a balance of one. The lower and upper limits are changed to zero and two respectively. The notify action will now only notify the user if the account is completely depleted, in other words, if the balance is zero. These changes are performed in order to illustrate the conformance check in a concise manner.

Figure 4.3 shows the original model of the bank account S and two mutated versions M and N. The number in the state label represents the amount that is on the bank account. s_0, m_0, n_0 are the states in which the bank account is depleted. The initial states of the LTS are s_1, m_1, n_1. The transitions $?d$ and $?w$ stand for the deposit and withdraw actions. They are input actions. The only output action possible is $!n$, which stands for notify.

All input actions are possible in every state. The states that did not have all input actions were modified. For example, $?d$ was added to the state m_1 as a self-loop. In order to make the implementations input-enabled, these transitions were added to the mutants M and N. In addition, quiescence edges δ were added to the mutants. They are marked in blue color. A quiescent state is a state that has no edge labeled with an output or an internal action.

The first mutant added the input action $?x$ marked in red. The action $?x$ does not exist in the specification at all. The states m_0 and m_2 need self-loops with the new action to remain input-enabled. This is however omitted in the graph to keep the example readable. The action $?x$ results in an arbitrary additional state since it is not relevant for the example. From the ioco definition, only traces that exist in the specification are checked. Since the only difference is the added action $?x$, M ioco S. The mutant cannot be detected or in other words is not viewed as an error.

The second mutant added the withdraw action to the depleted state resulting in a full bank account. Again, the mutated transition is marked in red. This time $N \cancel{ioco} S$. We can see the violation if we analyze the trace $?w?w$.

$$\begin{aligned}
&out(n_1 \quad after \quad ?w?w) = \{!n, q\} \\
&out(s_1 \quad after \quad ?w?w) = \{!n\}
\end{aligned}$$

Since $\{!n, q\} \not\subset \{!n\}$ the mutant was detected and is viewed as an error. The trace $?w?w$ can be further processed to generate an abstract test case.

4.3.2 Refinement Checking

In addition to the previously described ioco check a refinement check is performed. The refinement check for AS is strict but efficient. The refinement check is performed before the ioco check. Only if non-refinement is reached an ioco check is performed. The ioco check starts from the unsafe state and is then applied to observe if the mutation propagates to an unspecified output action. This way the expensive process of ioco checking is limited to a partial model, allowing higher exploration depths and hence longer test cases [61].

The following definitions are mainly based on Aichernig and Jöbstl [4].

Refinement checking is performed to find an unsafe state. A pre-state $\overline{v}$ is called unsafe if it shows not conforming behavior in a mutated model M^M with respect to an original model M^O. We refer to this unsafe state as u for unsafe, which is formally described as follows:

$$u \in \{s | \exists s' : M^M(s, s') \wedge \neg M^O(s, s')\}$$

The inputs for the refinement check are the original system and the mutated model. As stated before, the systems consist of a set of actions. They are combined with a non-deterministic choice operator. In the AS event traces and system states before $(\overline{v}, tr)$ and after one execution step $(\overline{v}', tr')$ are observed. $\overline{v}$ refers to the pre-state of variables and $\overline{v}'$ refers to post-state of variables. The mutated version AS^M refines to its original version AS^O iff all observations possible in the mutated version are possible in the original. Refinement is based on event traces and states. Furthermore, reachability is taken into account since not all states are reachable from the initial state [6]. Refinement of action systems is defined as follows:

$$AS^O \sqsubseteq AS^M =_{df} \forall \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr) \wedge P^M) \Rightarrow P^O$$

P^O and P^M refer to the do-od blocks of their respective action systems AS^M and AS^O. We negate the refinement definition and consider that actions A_i are selected in a non-deterministic choice. This allows us to obtain a set of constraints for detecting non-refinement. Non-refinement for ASs is defined as follows:

$$AS^O \not\sqsubseteq AS^M \textbf{iff} \bigvee_{i=1}^{n} \forall \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr) \wedge A_i^M \wedge \neg A_1^O \wedge ... \wedge \neg A_m^O)$$

These non-refinement constraints are given to a constraint solver to check whether it can be satisfied by some $\overline{v}, \overline{v}'$ for A_i with its parameter. It is sufficient to satisfy only one of the constraints in order to find non-conformance. Therefore, the actions are checked one-by-one until a non-refinement constraint is satisfied, which means an unsafe state was found. If the unsafe state is also actually reachable we can then proceed with the previously discussed ioco check. If no unsafe state was found we know that the ASs are equivalent and no further ioco check is needed.

As stated before the back-end is written in SICStus Prolog which has an integrated constraint solver clpfd (Constraint Logic Programming over Finite Domains) [28]. This was the initial constraint solver for MoMuT. The newer version of MoMuT in this thesis uses the Microsoft's Z3 constraint solver instead.

5 Using Rule-Engine Models in Model-Based Testing

In order to perform model-based testing (MBT) a rule-engine model (REM) is translated into an extended finite-state machine (EFSM). EFSMs were chosen as a representation since they are well-studied [46] and are easy to translate into EFSMs. The translation process is shown in Section 5.1. In Section 5.2 a property-based testing approach is shown that utilizes said EFSMs for MBT. Figure 5.1 shows a flow chart of the proposed approach. In this chapter, the processes and data nodes in the upper-most row are described. How to integrate and implement the external test case generator is shown in Chapter 6. The content of this chapter is based on Aichernig and Schumi [9]. This is explained in more detail in the introduction of this thesis in Section 1.3.

5.1 Translating Business Rule Models into EFSMs

This section introduces the syntax of the REMs as well as the definition of EFSMs that is used in this thesis. Finally, the transformation process from REM to EFSM is explained. The term REM is used as a synonym for business rule model introduced in Chapter 2. In the context of the case study, REMs describe the functional model of the system. The model focuses on rules that constrain the business domain. Therefore, the preferred term will be REM, to emphasize on the models of our case study.

5.1.1 Rule-Engine Models

The web-service application of the case study has a custom rule management system. At the time the tool was developed many approaches to embed rules into a system that exist today were not developed or established yet. The implementation is similar to other standardized rule definitions. Rule management systems and their respective tools are explained in more detail in Section 2.3. The approach presented in this section can be adjusted and transferred to other rule management systems.

The approach focuses on REM. A REM describes the behavior of an object in its respective domain. The object has an identifier, a state, and attributes. The REMs have a specific custom syntax in order to comply to the SUT in the case study. They are contained in XML files. The syntax of a REM can be defined as in Figure 5.2.

It is well-known in the studies of compiler construction, that proper definition of syntax is fundamental to automated processing of the code. Unfortunately, this is still often neglected

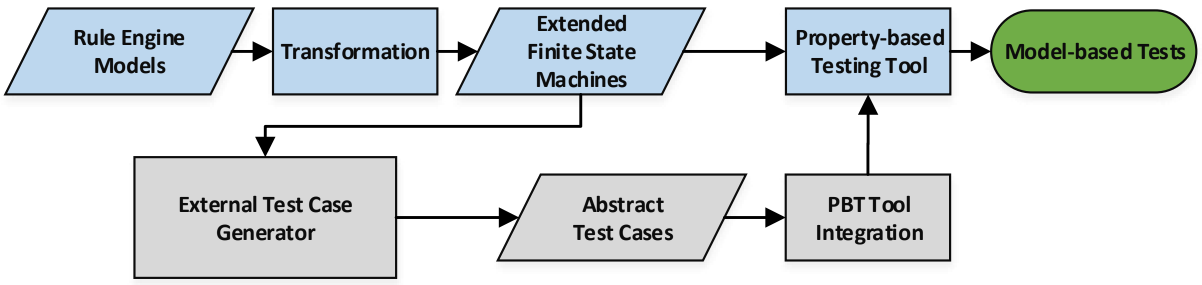

Figure 5.1: Flowchart of the proposed testing approach showing how REM can be testing using a PBT tool.

$$
\begin{aligned}
REM &=_{df} rem(AllAttributes, AllTasks, AllStates) \\
AllAttributes &=_{df} Attribute* \\
Attribute &=_{df} attr(id : Name, type : DataType, params : Parameter*) \\
DataType &=_{df} Integer|Float|Bool|String|Enum|Object|Date|DateTime| \\
&\quad TimeSpan|File|Reference|... \\
Parameter &=_{df} MinValue|MaxValue|EnumItem * |Query|Regex|... \\
\\
AllTasks &=_{df} Task* \\
Task &=_{df} task(id : Name, possibleNextStates : Name*, \\
&\quad attributes : DynamicAttributeInfo*) \\
DynamicAttributeInfo &=_{df} (attribute : Name, enabled : bool, required : bool) \\
\\
AllStates &=_{df} State* \\
State &=_{df} state(id : Name, possibleTasks : Name*)
\end{aligned}
$$

Figure 5.2: Definition of the custom REM syntax based on Aichernig and Schumi [9].

in today's economy. In the study, the syntax limits the scope of what a REM is allowed to contain and therefore what the test framework is supporting. Behavior deviating from the defined syntax is not supported. New features for REMs have to be carefully implemented in order to comply with the abstract syntax or to only cause minimal changes. The abstract syntax was defined in order to create a XML-schema following the guidelines of Thompson et al. [102].

There exist various tools to transform XML files into source code using schemas. Since the case study was performed in C#, Microsoft's XML schema definition tool[1] (xsd.exe) was used to create the source code. Note that this step can be performed in most other object-oriented programming languages with their respective tools.

A REM consist of a set of attributes, a set of tasks and a set of states. An attribute represents data that has to be sent to the SUT in order to perform a task. An attribute comprises a name, a data type and optionally a set of parameters. The data type describes which type of data is saved in the attribute. This can range from a simple type such as integer to more complex data types such as object types which are a composition of other attributes. Parameters can be supplied to further restrict the data type. For numeric values, this may be the minimum and maximum value of the integer, for strings this may be the minimum and maximum length of the string. More complex restrictions can occur in the form of regular expressions which are typically implemented for validating user input. Attributes may also be restricted via queries. This allows the implementation of a dynamic drop-down menu in a web-form where the query results are presented to a user, e.g. for the selection of the responsible person for a certain issue.

Tasks represent the actions or events a user may trigger. They are the transition relation and can be represented by multiple transitions in a state machine. A task consists of a name, a set of possible next states and a set of dynamic attribute information. As the name suggests, the set of possible next states describes the states the object may transition into. If more than one state is possible, the user may choose which state the object should be transitioned into.

[1]https://msdn.microsoft.com/en-us/library/x6c1kb0s.aspx

```
 1 <RuleEngineModel TfmsType="Incident">
 2   <AllAttributes>
 3     <StaticAttributeInfo Name="Name" DataType="String" MinValue="1" MaxValue="64"/>
 4         <StaticAttributeInfo Name="Severity" DataType="Enum">
 5       <EnumItems>
 6         <EnumItem Name="low" MlgKey="Incident.Severity.low" /> ...
 7       </EnumItems>
 8     </StaticAttributeInfo>
 9         ...
10   </AllAttributes>
11   <AllTasks>
12     <Task Name="IncidentCreateTask">
13       <DynamicAttributesInfo>
14             <Attribute Name="Name" Enabled="true" Required="true" />
15                 <Attribute Name="Severity" Enabled="true" /> ...
16       </DynamicAttributesInfo>
17       <PossibleNextStates>
18         <State Name="Submitted" NoteRequired="false" />
19       </PossibleNextStates> ...
20           <RequiredUserRoles>...</RequiredUserRoles>
21     </Task>
22   </AllTasks>
23   <AllStates>
24     <State Name="Submitted" MlgKey="Incident.State.Submitted">
25       <PossibleTasks>
26         <Task>IncidentEditTask</Task> ...
27       </PossibleTasks>
28     </State> ...
29         </AllStates> ...
30 </RuleEngineModel>
```

Listing 5.1: Simplified XML representation of a rule-engine model based on Aichernig and Schumi [9].

While the user makes this choice deliberately on the system from a testing perspective we do not know which choice the user will make and assume that every choice is as likely. The set of dynamic attribute information contains the attributes associated with the task. Some attributes are required in order to perform the task and others are optional and hold additional information. Furthermore, an attribute may be disabled, which means the user is not allowed to edit the given attribute in the current task.

Listing 5.1 shows a simplified version of the incident manager's (ICM) REM encoded in XML. The ICM is the simplest module consisting of only one REM and is therefore shown throughout this chapter as an example.

The relevant sections for our approach are:

- attribute definitions with data types and parameters (Line 2 - 10)
- transitions with the dynamic attribute information, possible next states and required user privileges (Line 12 - 22)
- states with their possible tasks (Line 23 - 29)

In addition to the main components, models can also have scripts which are executed on certain events, queries for the selection of objects, and reports for an overview of the entered objects.

5.1.2 Extended Finite-State Machines

In the following, syntax definitions and translation functions will contain operators which bear special meaning. Some of the notations presented here are based on Bjørner and Jones [23].

$\{a, b, c\}$. Set enumeration: the set of elements a, b, and c.
(a, b, c). Tuple enumeration: the tuple of elements a, b, and c.
$[a, b, c]$. Sequence enumeration: the sequence of elements a, b, and c.
$s1\hat{}s2$. The sequence formed by concatenating sequences $s1$ and $s2$.

Furthermore, lower-case variables will usually describe single elements while upper-case variables will describe a collection of elements.

EFSMs have found use in the software testing community [108, 29]. Since REMs are easy to translate into EFSMs and EFSMs are also useful in model-based testing they were chosen as an intermediate model in this study. An EFSM is a very general concept and it is easy to infer model-based tests from it.

EFSMs have been differently introduced in the literature. Cheng and Krishnakumar [32] introduced them as a 7-tuple and Li and Wong [72] as a 5-tuple. The definition in this thesis is based on Ramalingom et al. [89].

An EFSM is defined as a 6-tuple $M = (S, s_0, I, O, T, V)$, where

S is a nonempty set of states,
s_0 is the initial state of the EFSM,
I is a nonempty set of input interactions,
O is a nonempty set of formal output interactions,
T is a nonempty set of transitions,
V is a set of variables.

Furthermore, a *Transition* is defined as a 5-tuple $t \in T = (s, d, i, p, cb)$, where

$s \in S$ is the source state,
$d \in S$ is the destination state,
i is an input interaction from I or empty,
p is a predicate or a guard,
cb is a compute block which consists of assignment statements or output interactions $o \in O$.

An EFSM model is a generalization of the traditional state machine model. In contrast to the state machine model where the model represents the entire state explicitly, the EFSM represents a set of states and a bounded range for each data register. In addition to the regular state machine behavior, the EFSM has enabling functions (guards) that are associated with each transition and arithmetic functions that perform data operations in the form of output signals [33].

Figure 5.3 shows our reoccurring bank account as an EFSM. We added the log-in mechanism to have more states and removed the deposit action to reduce the number of transition and keep the example concise. The flat circles represent the states S and *Start* is our initial state. The text above a transition contains the input interactions i and the predicate p. Output interactions o and assignment statements can be found below the line.

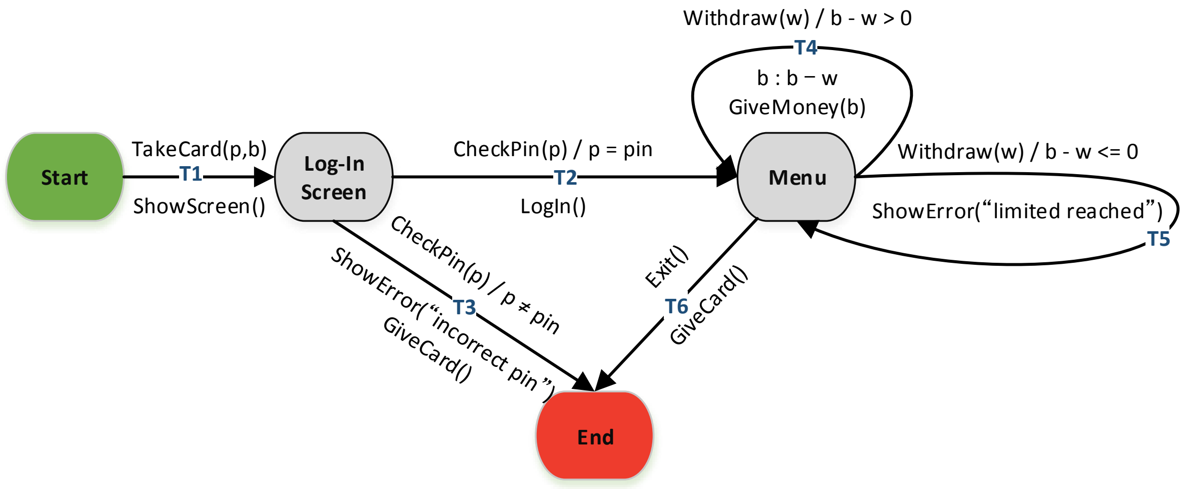

Figure 5.3: Representing a bank account log-in as an EFSM.

5.1.3 Translation Function

In the previous sections the syntax definition of the REM was introduced and the EFSM definition for this thesis was shown. In order to translate between those two structures, a transformation function has to be executed. The transformation process is shown in Figure 5.4.

Some of the sets in the *EFSM* definition can be translated straightforward from the *REM* syntax. For example, the set of states S is generated by using the name of each state in *AllStates*. The set of input interactions I, the set of formal output interactions O and the set of variables V is translated in a similar manner by using their respective sets *AllTasks*, *AllStates* and *AllAttributes*. Since the name of the elements is unique in each set, it is enough to identify each object safely. The initial state S_0 cannot be translated from the *REM* since there is no definition for it. Therefore, the constant string "Global" was chosen.

The last set, the set of transitions T, needs to be derived from the *REM* syntax since there is no straightforward translation function. T consists of 5-tuples t of the form (s, d, i, p, cb). The state s is taken of an element of the set of states S. In a state certain tasks are possible (possibleTasks). From these tasks, a next state is selected. The name of this state is the destination state d. The input interaction i is a combination of the task name and the destination. The operator ˆ symbolizes this string concatenation. By combining these two strings ($task.Name\hat{}d$) it is ensured that a transition is unique and identifiable and thus prevents non-determinism. The predicate p is always enabled. This simplification can be realized because the only tasks considered are the ones that are possible in the current state.

The last part of the 5-tuple is the compute block cb. The output of the model contains the destination state and also assigns the attributes of the task. This is done by using the type and the parameters of the attribute. They are passed to a generator which then produces an arbitrary value satisfying the type and parameter constraints. The generation of attributes is explained in the next section. It is solely shown in the translation step already since the two procedures are not entirely separable and preparation steps are therefore performed.

The result of the translation process can be seen in Figure 5.5. The class diagram consists of classes that together build the model. The *ModelEFSM* is the model that is translated from a REM. In the case study, each module of the system consists of multiple REMs. Therefore, they are collected in the *ModelModule*. This is explained in more detail in Section 5.2.2. The *ModelEFSM* class itself holds the initial state s_0 and the states S in a Hashset. The set of transitions T is represented by a dictionary mapping from the transition names to transition objects. Also, attributes are separated from the model and accessed in the same manner.

$$
\begin{aligned}
&\mathit{buildREM} : \mathit{REM}(\mathit{AllAttributes}, \mathit{AllTasks}, \mathit{AllStates}) \rightarrow \mathit{EFSM}(S, s_0, I, O, T, V),\ \mathit{where}\\
&\quad S = \{s.\mathit{Name} \mid s \in \mathit{AllStates}\},\\
&\quad s_0 = \text{"}\mathit{Global}\text{"},\\
&\quad I = \{t.\mathit{Name} \mid t \in \mathit{AllTasks}\},\\
&\quad O = \{s.\mathit{Name} \mid s \in \mathit{AllStates}\},\\
&\quad T = \mathit{buildT}(\mathit{AllAttributes}, \mathit{AllTasks}, \mathit{AllStates}),\\
&\quad V = \{a.\mathit{Name} \mid a \in \mathit{AllAttributes}\}.
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{buildT} : \mathit{AllAttributes} \times \mathit{AllTasks} \times \mathit{AllStates} \rightarrow T, \mathit{where}\\
&\quad T = \{(s, d, i, p, cb) \mid\\
&\qquad s = \ \mathit{state.Name} \ \wedge \ \mathit{state} \in \mathit{AllStates} \ \wedge\\
&\qquad d = \ \mathit{dest.Name} \ \wedge \ \mathit{task} \in \mathit{state.possibleTasks} \ \wedge \mathit{dest} \ \in \mathit{task.nextStates} \ \wedge\\
&\qquad i = \ \mathit{task.Name\hat{}d} \ \wedge\\
&\qquad p = \ \mathit{true} \ \wedge\\
&\qquad cb = \ (a_1 = g_1.\mathit{value}, ..., a_n = g_n.\mathit{value}, \mathit{output} = d) \ \wedge\\
&\qquad\quad a_i \in \mathit{task.attributes} \ \wedge \ g_i = \mathit{generator}(a_i.\mathit{type}, a_i.\mathit{params})\}.
\end{aligned}
$$

Figure 5.4: Translation function of a REM to a 6-tuple EFSM based on Aichernig and Schumi [8].

While transitions can be dynamically generated for the individual transitions, a class for each attribute type with their respective generators had to be created.

5.2 Using EFSMs for Property-Based Testing

In order to implement EFSMs for PBT, a tool has to support certain functionality. In this section, a way of using EFSM in PBT and how to classify the PBT tool's functionality into components is explained. This is by no means functionality a PBT tool has to have. It can be seen as functionality supported by a PBT tool which enables the implementation of the approach presented in this thesis.

Test cases have to be constructed as a sequence of operations. It has to be ensured that each test case finishes in a clean state, side-effects of one test case should not affect the outcome of other test cases. The representation of test cases should be easy to process. For example, QuickCheck has a symbolic test case representation which makes it easy to store the test cases, collect statistics and most importantly allow to write shrink functions [52]. It has to be possible to apply generators to generate different types of operations. An operation ideally supports four components named

1. *Precondition*, which verifies if the operation is appropriate in the current state,
2. *Run Model*, which executes the operation on the specification of the system,
3. *Run SUT*, which exercises the system,
4. *Postcondition*, which checks that the result of each operation satisfies the property.

These four components are needed in order to perform the approach shown in this thesis. However, some of the PBT tools may logically structure the functionality differently. For the

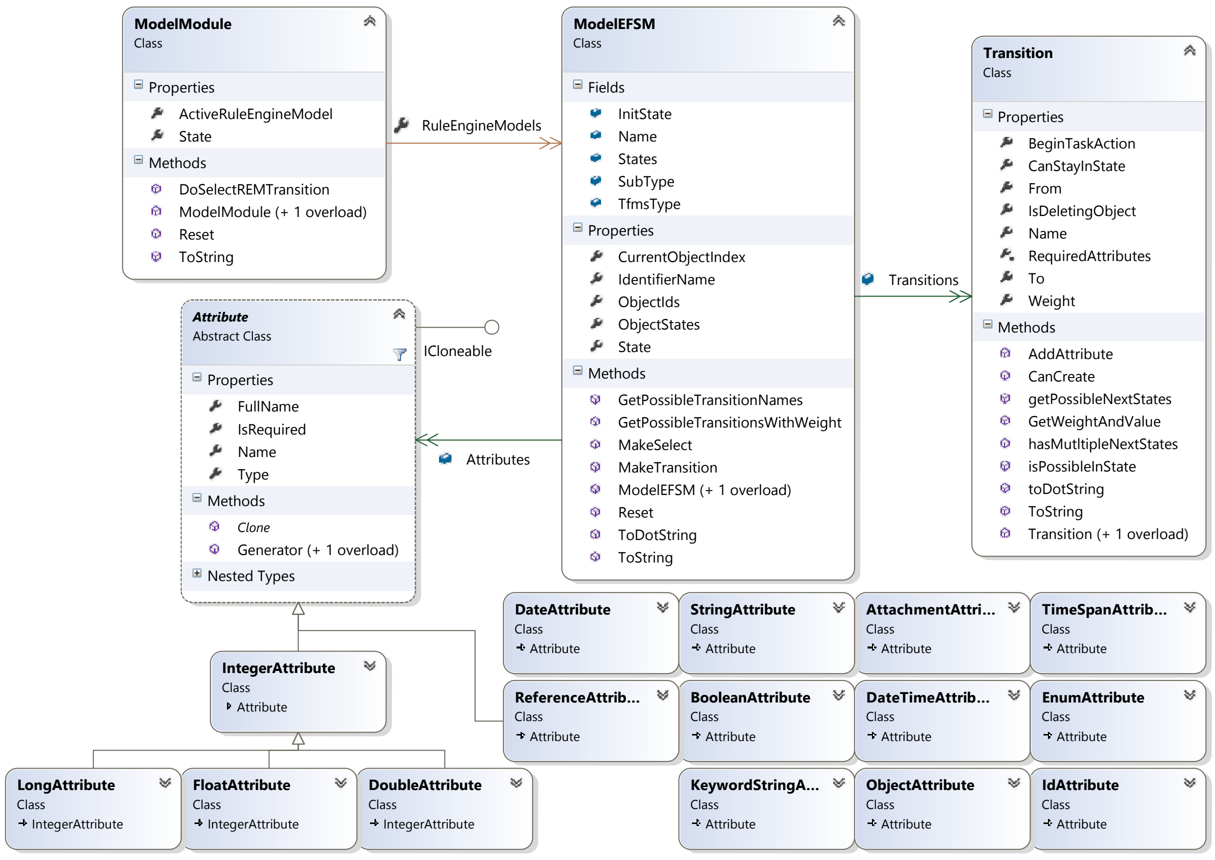

Figure 5.5: Class diagram of the translated model.

case study FsCheck's experimental version was used. There, an operation has three functions. Which are *Pre*, *Run*, and *Check*. *Pre* is equal to *Precondition*. Run is namely *RunModel*. *Check* exercises the SUT and returns a result encoded as a property. Namely, this is *RunSUT* and *Postcondition* in one functional block. In the stable version of FsCheck, the functionality is encoded in four functions and the operations are called commands. Many other PBT tools also support the needed behavior but similar as in the experimental version, the functionality is structured logically differently.

Before defining a property we need to introduce definitions for the *Model*, the *SUT* and the operations themselves. The following definitions are based on Aichernig and Schumi [8]. As the definitions were introduced for the commands of the stable version they were adapted in this thesis to fit the operations of the experimental FsCheck version.

A *Model* is defined as a 3-tuple $model =_{df} (s, T, doStep : i \rightarrow Model)$, where

s is the current state of the model,
T is a nonempty set of transitions from the EFSM definition,
$doStep$ is a function that executes a transition $t \in T$ based on the input i and returns a new *Model* instance $model'$.

A SUT is defined in the same manner as the model. A *SUT* is a 3-tuple $sut =_{df} (s, T, doStep : i \rightarrow SUT)$, where

s is the current state of the SUT,
T is a nonempty set of transitions from the EFSM definition,
$doStep$ is a function that executes a transition $t \in T$ based on the input and returns a new *SUT* instance sut'.

$$\begin{aligned} operation =_{df} \quad & op(i : Input, Pre : Model \rightarrow bool, Run : Model \rightarrow Model, \\ & Check : Model \times SUT \rightarrow bool) \quad where, \end{aligned}$$

$$\begin{aligned} op.Pre(model) =_{df} \quad & \begin{cases} \text{True}, & \text{if} \quad t \in T : t.s = model.s \\ \text{False}, & \text{otherwise} \end{cases} \\ op.Run(model) =_{df} \quad & model.doStep(op.i) \\ op.Check(model, sut) =_{df} \quad & \begin{cases} \text{True}, & \text{if} \quad sut' = sut.doStep(op.i) \wedge model.s = sut'.s \\ \text{False}, & \text{otherwise} \end{cases} \end{aligned}$$

Figure 5.6: Formal definition of an operation.

$$\begin{aligned} property =_{df} & \forall op \in OP : \\ & op.Pre \implies model' = model.doStep(i) \wedge op.Check(model', sut) \end{aligned}$$

Figure 5.7: Formal definition of a property for an EFSM.

Note that the function *doStep* returns a new instance of an SUT. In real use cases, we often do not need to return a new instance of the SUT as the *doStep* function will update the state *s* of the SUT. In comparison to that, we never update the state of the model when we perform *doStep*, but rather return a new instance of a model with a new state. Or in other words, the SUT might be mutable while the model must be immutable. The model is required to be immutable by FsCheck as the framework saves the model instances after each operation and uses them later when verifying the operations.

A definition for an operation is shown in Listing 5.6. An operation has an *Input* which is used to create the specific type of operation. Additionally it has the previously explained functions *Pre*, *Run*, and *Check*.

5.2.1 Property of an EFSM

The property of an EFSM can be encoded as follows. For each permitted path on the model, the postcondition of each transition must hold. In our case study, we chose to compare the current state of the SUT with the model as a postcondition. However, this can be extended to not only compare the state between model and SUT but also compare if values assigned to the attributes are correct.

The definition of a property is based on Aichernig and Schumi [8] and shown in Figure 5.7.

The precondition of the operation checks the state and input of the corresponding transitions to verify if the operation is valid in the given context. In our case-study the generators only create valid transitions to save generation time. One could assume that the precondition can therefore always be set to true. However, shrinking only works if preconditions are supplied. Shrinking deletes operations from a test case, which means that operations following the deleted operation may be executed from a different state they were generated in. This can result in operations being executed in an invalid state. Therefore, it might be useful to still specify the guard of the transition to check the current state to avoid wrongful shrinking. Both *RunSUT* and *RunModel* perform a computation on their respective parts based on the input and state and they produce output interactions. These output interactions are compared in

```
public override Gen<object> Generator(int? minValue, int? maxValue)
{
  if (minValue.HasValue) {
    if (maxValue.HasValue) {
      return Gen.Choose(minValue.Value, maxValue.Value).Select(l => (object)l);
    }
    else {
      return Gen.Choose(minValue.Value, int.MaxValue).Select(l => (object)l);
    }
  }
  if (maxValue.HasValue) {
    return Gen.Choose(int.MinValue, maxValue.Value).Select(l => (object)l);
  }
  return Arb.Default.Int32().Generator.Select(l => (object)l);
}
```

Listing 5.2: Integer generator derived from REM.

the postcondition of the operations. In our case, the current state of the model is compared with the current state of the SUT. If they are the same, the property of the operations holds, if not the property of the operation, and furthermore of the whole EFSM does not hold. If the property holds for all operations of the test case, the property of the EFSM holds.

5.2.2 Integrating EFSM into FsCheck

In this section, the integration of our previously defined EFSM into FsCheck is shown. FsCheck is merely one of many PBT tools. It should be possible to apply the presented approach into other PBT tools and thereby other programming languages. However, it is important to show the approach with a tangible example and not solely abstract.

At the core of a PBT testing tool, we operate with generators. Therefore, one of the first steps is to define them. Each attribute has to be provided with a value when performing a task on the SUT. These attributes have their type embedded in the REM. This type and other restrictive parameters define the generators. A simple integer generator is shown in Listing 5.2. Whenever an attribute of the type integer is used in a task, its value will be generated by this generator.

At the core of this generator, we use the function *Gen.choose* to choose an integer value between a lower and upper bound. This is common for all types of generators. An integer attribute may have restrictions for maximum and minimum values. These restrictions are represented through the conditionals and the lower and upper bounds of the *Gen.choose* function. To be used dynamically all our generators are of the type object. We have to specifically cast the generator value from integer to object before returning.

Other attributes cannot be generated in a straightforward manner as shown above. String attributes that are restricted by regular expressions have to be generated with a different approach since the computation time would be too expensive. This was explained in more detail in Section 3.3.1. A more dynamic approach is possible with the help of Fare[2], a .NET port for Google's java library Xeger[3]. The tool uses a finite-state automata implementation in order to generate text that matches a certain regular expression. This external string generation

[2]https://github.com/moodmosaic/Fare
[3]https://code.google.com/archive/p/xeger/

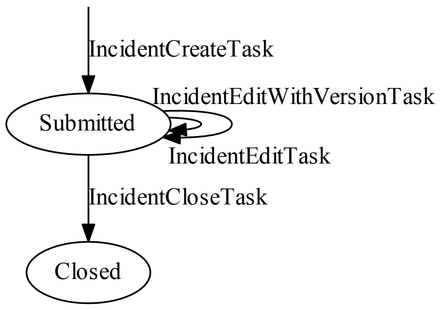

Figure 5.8: Possible transitions for an incident object.

```
public static Gen<Operation<IModule, ModelEFSM>> Next(ModelEFSM m) {
  IEnumerable<WeightAndValue<Gen<Transition>>> wvs;
  wvs = m.ActiveRuleEngineModel.GetPossibleTransitionsWithWeight();

  return Gen.Frequency(wvs).SelectMany(t => {
    IEnumerable<string> states = t.PossibleNextStates(m.ActiveRuleEngineModel.State);
    return Gen.Elements(states).SelectMany(nextState =>
      GenerateData(t.RequiredAttributes.Values)
      .Select(attributeData => (Operation<IModule, ModelEFSM>)
      new DynamicOperation(t.Name, attributeData, nextState)));
  });
}
```

Listing 5.3: Next implementation using a dynamic operation generator.

strategy was adjusted in this thesis to supplement the generators of FsCheck.

Object attributes and reference attributes rely on other elements. An object attribute, for example, is an attribute that has other attributes as children and is accessed via list operators. Since the structure of the REM is flat and not nested it has to be parsed multiple times. This behavior is similar to a multi-pass compiler.

To illustrate the following functionality we will make some simplifications to our model. In our case study, we present the results of our approach with more complex models. Here, we will use the smallest module available from our industry partner AVL, the incident manager module (ICM). This module consists of only one REM. Therefore, no switching between REMs has to be performed. We further simplify our model by assuming there is only one incident object processed throughout our test case. The state machine in Figure 5.8 represents the possible tasks of one incident object.

A task in the REM is equivalent to a transition in the EFSM or an operation in the test case of our PBT tool. As explained in Chapter 3, the user has to supply the *Next* function which returns a generator for the operations. The generator is shown in Listing 5.3.

First, a set of generators is chosen which contains only possible transitions from the current state. They are stored in *wvs* (WeightedValues). Out of those one transition generator is chosen to take into account the weights of each generator. For this transition, all possible next states are considered and one is chosen. Finally, the required attributes for the transition are generated with the previously explained principle. With this information, we are able to generate a dynamic operation which transitions the SUT and model from a source state to a

```
1  public class DynamicOperation : Operation<IModule, ModelEFSM> {
2    string TransitionName { get; set; }
3    string NextState { get; set; }
4    Dictionary<string, object> AttributeData { get; set; }
5
6    public DynamicOperation(string transition,
7    Dictionary<string, object> attributeData, string nextState) {
8      TransitionName = transition;
9      NextState = nextState;
10     AttributeData = attributeData;
11   }
12
13   public override bool Pre(ModelEFSM m) {
14     Transition transition = m.ActiveRuleEngineModel.Transitions[TransitionName];
15     var isValid = transition.From.Contains(m.State) ||
16                   transition.From.Contains(Const.GlobalState);
17     return isValid;
18   }
19   public override ModelEFSM Run(ModelEFSM m) {
20     var n = new ModelEFSM(m);
21     n.ActiveRuleEngineModel.MakeTransition(TransitionName,
22       NextState, n.CurrentObjectIndex);
23     return n;
24   }
25   public override Property Check(IModule a, ModelEFSM m) {
26     Transition transition = m.ActiveRuleEngineModel.Transitions[TransitionName];
27     a.Task(m.ActiveRuleEngineModel, transition, AttributeData, NextState);
28     var hasSameState = a.State.Equals(m.State);
29     return hasSameState.ToProperty();
30   }
31   public override string ToString() { return TransitionName; }
32 }
```

Listing 5.4: Implementation of a dynamic operation.

destination state. Note that only possible transitions and states are considered.

In order to handle the execution of transitions, dynamic operations are implemented. A dynamic operation supports all types of transitions of the EFSM and is shown in Listing 5.4. The previously introduced generator function returns an instance of this dynamic operation.

The transition, the next state, and the attributed data are passed into the operation as constructor arguments (Lines 6-11). The precondition (Lines 13-18) verifies if the current operation is valid in the context. As mentioned before only valid transitions are chosen. For regular test execution, the precondition can be omitted. However, if shrinking is enabled the precondition has to be implemented to ensure that no invalid sequences are created by deleting wrong operations.

In *Run* (Lines 19-24) the transition is executed on the model. Note that the new model is a deep copy of the old one. FsCheck will store the returned model of the function later to test the postcondition. If only one model object is created throughout the run, following operations will alter the model object resulting in incorrect checks for the preceding operations. A short example will explain this problem. Assume a model with two states s_0 and s_1 and transitions that t_0 and t_1 which can be executed from any state and will set the model state to their respective numbers. We further assume the test sequence t_0, t_1. If we only create one model object throughout the test run the state after t_0 will be s_0. Then, the next transition t_1 is executed and the model state is changed to s_1. Since all operations are run on the model before the first one is checked the model state is already changed to s_1 when checked in the first operation. It should be s_0. It is in general better to create value types over reference types for these models if possible. Due to the complexity of the case study, this was not possible in the context of the thesis.

The operation is executed on the actual SUT (Line 26-27). The *IModule* class represents the interface to the SUT. It executes the corresponding task on the SUT and also sets the attributes of the task. This is done via reflections. Finally, the state of the model is compared with the state of the SUT and then returned as a property (Line 28-29). In this section, it would also be possible to compare the values of the previously set attributes to ensure they were set properly. This would strengthen our postcondition. The *ToString* function is overwritten to support pretty printing of operation sequences.

Embedding the logic of transitions in a dynamic operation rather than individual operations with their respective attributed data reduces the amount of manual coding. Once the transformation steps from the REM to the EFSM and the integration in FsCheck is hooked up the dynamic operation is able to handle most transitions needed in a test case. However, as in most real examples, there are some exceptions. Some operations create new objects such as the *IncidentCreateTask*. In order to handle multiple objects of a module at the same time, the state of each module has to be saved and the currently active object has to be remembered by the model. In the application, an object can be seen as a new tab in a web-form. It has the same type of input fields, but the specific provided data is different. Multiple of those tabs can be open at the same time. In order to switch between those objects, a *SelectOperation* was added to the generator functions. If this operation is executed the currently active object of the REM is switched. The state and the id of the object are compared between model and SUT.

As mentioned before, in the case of the ICM the module consists of only one REM. Other modules usually contain multiple REMs. In order to allow multiple REMs simultaneously in a test case, a similar principle to the *SelectOperation* was implemented. Instead of switching between objects of one REM, we switch the whole REM with another one. This is the called *SelectREMOperation*. The postcondition for this operation is the comparison of the current object state of the currently active REM. It is straightforward to take this concept of selecting different items at a given layer to the next layer by switching between whole modules. However, it was not implemented during the case study since the focus was on testing the modules separately. Test cases would become verbose and more time-consuming.

Those two artificially created operations are also included in the operation generator. The operations are weighted in relation to the maximum amount of possible transitions. This is done to ensure that the select operations are not executed too often. The operation generator shown before in this chapter can be adjusted to not only create dynamic operations but also consider the select operations by using one of the generators with their respective weight. This can be done by using the following FsCheck command:

$Gen.OneOf(Gen.Frequency(gens))$, where gens is the set of generators.

5.2.3 Optional Attributes

So far only attributes that are required to perform a task were considered. Yet there are additional attributes that can be set when a task is performed. For an incident object, this can include the severity of the issue or additional documents to describe the issue further. In a small REM such as the ICM, there are about 50 different attributes. Bigger REMs contain up to 1000 attributes, while most of the attributes are optional. This means, testing optional attributes is a big part of testing the system. Fredlund et al. [45] used JSON objects that have optional keys and generated them using QuviQ QuickCheck. They generate a random natural number greater or equal to the number of required properties. First, the required properties are generated. Then, additional properties are generated and it is ensured that they are unique. They did not further specify with which underlying distribution the natural number is generated.

For our case study, we tried out the described approach. If the natural number is generated with a uniform distribution it is as likely that almost all optional attributes are added to a task as none. With a REM consisting of up to 1000 attributes, this resulted in tasks being very different from actual user behavior. In addition, the execution time of a test case increased significantly.

In order to adjust the number of additional attributes for each REM individually a more dynamic approach is needed. Every time a task is executed a *ChooseFunction* is called that takes a randomly generated value x between 0 and 1 as input to calculate its output y, which again has to be between 0 and 1, where y is the percentage of optional attributes to be included. This *ChooseFunction* is called in an *ElementSelector* to select n random unique attributes out of the set of all attributes. By implementing the *ChooseFunction* as an interface or an abstract class the ability to switch between different functions is given. For example, if an exponentiation function such as x^9 is chosen the expected amount of attributes per task is 10 percent. This can be easily calculated by solving the following basic integral:

$E(x) = \int_0^1 x^9 dx$

It is also possible to disable optional attributes by implementing a *ChooseFunction* which always returns 0. Using a *ZeroFunction* as a *ChooseFunction* is equal to the approach presented previously, only considering the required attributes. The distributions can be set individually for each test case or for each *REM* allowing proper configurations for smaller and bigger models as well as for test cases that focus on different aspects of the SUT. It is easy to extend this approach to other useful functions such as polynomial functions which are based on real data. Solely a new class has to be derived from *ChooseFunction* implementing a function transforming an x value to a y value, both between 0 and 1. Another interesting usage could be to deliberately choose generators for attributes that test negative behavior. In the case study, only values between the accepted boundaries are tested. However, it might be interesting to test how the system responds to invalid input. This could be done by generating values that are prohibited by the REM. Since this is not straightforward for all attributes this was out of the scope of this thesis but could be a logical next step of creating a richer model.

6 Integration of External Test-Case Generators

The transformation process of rule-engine models (REMs) to extended finite-state machines (EFSMs) and how to integrate them into a property-based testing (PBT) tool was shown in the previous chapter. This chapter will focus on how external test-case generators can be integrated into that process. This gives the tester more control on how to produce meaningful operation sequences for the test cases. First, the interface for external test-case generators is explained and how it can be integrated into a PBT tool. Then, the integration process is shown with a simple regex-based operation generator. Finally, the integration process is shown with MoMuT as a generator for abstract test-cases. The test cases generated from MoMuT were also used to test the system in the case study.

6.1 Interface for External Test-Case Generators

The idea of the presented approach is to define an external generator to create a sequence of operations that form a test case. A PBT tool like FsCheck uses a function *Next* to generate the next operation. That means the scope of the *Next* function focuses on one operation at a time. The operation that is currently generated relies on the current state of the model. The idea to generate one operation at a time leads us to only consider the current state of the model to generate the next operations. However, there are other data that can be utilized to generate a sequence of operations. Before executing any operation on the model, we can generate the sequence from an external source. This allows us to generate operations that not only rely on the current state. It is possible to also include the path up to the operation and even after the operation itself. This allows better control of the generation of the operation sequence since this process is decoupled from the PBT tool.

The simplest use-case of an external generator could be to replay test cases from a PBT tool. We generate a sequence from a PBT tool and save it externally. This sequence can then be reused as an external source and supplied to the PBT tool via the interface. This means we are effectively replaying the previously run test case. This use-case was discovered as a by-product of the thesis since it was very useful to try to reproduce errors or to create very specific test cases. *FsCheck* already supplies some sort of replay functionality. It is possible to set the seed of the random generator and make the test sequence generation deterministic. That way it is possible to replay a certain test case over again. However, as soon as the model changes or multiple test cases need to be replayed, or just some parts of the sequence should be replayed, this approach does not work anymore. Here, the replay functionality via an external generator showed its usefulness since it could cover all those cases. It proved to be incredibly useful for regression testing of the test framework itself. If any changes on the test framework were implemented, short test cases were written in order to target those. This usually resulted in finding mistakes earlier as compared to waiting until the generators produce a sequence that operates in the area that was changed.

6.1.1 Interface Design

In order to inject an external sequence in *FsCheck*, an interface to the tool had to be built. Note that interface does not refer to the programming structure used in object-oriented languages (where interface describes a set of methods that an instance of a class has without defining the semantics of the methods). Here, interface refers to a shared boundary between two components, namely the PBT tool and the external generator, to exchange the information from both components. First, we define our external generator gen_{ex} as a set of operation arguments:

$$gen_{ex} =_{df} OperationArg*$$

OperationArg can be of any type. It is up to the specific implementation of the external generator how and of what type the set of operation arguments is generated. Note that *OperationArg* is deliberately defined as generic as possible. While the arguments may contain only the type of the operation, they can also partly or fully supply all the information necessary to perform an operation. For example, the attributes of the REM can also be supplied from the external generator. In our case study, we supplied only the type of the operation. Based on the type, we select one of the internal generators introduced in the last chapter to generate the attributes.

The structure of a test case in PBT tools can be defined as shown in Figure 6.1. A test case consists of a sequence of operations. Operations were already defined in Section 5.2 of the previous chapter. For our interface, we extend the previously defined operations by adding a generated data to the operation. The external generator generates data based on the type of operation. The data can be used in *Pre*, *Run*, and *Check*. Note that data is deliberately defined as generic as possible. While data can contain only the type of the operation, it can also partly or fully supply all the information necessary to perform an operation. For example, the attributes of the REM can also be supplied from the external generator. In our case study, we supply only the type of the operation. Based on the type, we select one of the generators introduced in the last chapter to generate the attributes. Figure 6.2 shows a test case with an external generator.

Traditionally, the length of a test case is decided by the PBT tool. Based on the current state S an operation op_{exi} is then generated choosing the type of operation and the data of the operation with generators. In the presented approach the external generator is responsible for the length of the sequence. That means that in order to apply this approach it must be possible to control the length of an operation sequence. The type of operation is also defined by the external generator. The information needed for an operation can be supplied by either the external generator as the *OperationArg*, a regular PBT generator or a combination of both. For example, in the case study, the external generator decides on the types of operations and on the length of the operation sequence. A PBT tool then generates the attributes for each operation.

6.1.2 Interface Implementation

The interface was implemented by extending the abstract class *Machine* of FsCheck with new functionality. An external machine was derived from the machine which implements the new functionality. Figure 6.3 shows the new components and how they are wired together with the regular FsCheck parts. In FsCheck the *Machine* executes the operations on the model and the SUT. FsCheck uses a *Runner*, which is responsible to check the machine and reports the test results. The machine has to be transformed into a property to be checked. This functionality is

$$\begin{aligned} Testcase &=_{df} Operation_{ex}*, \quad where \\ Operation_{ex} &=_{df} op(arg_{ex} : OperationArg, Pre : Model \to bool, \\ &\qquad Run : Model \to Model, Check : Model \times SUT \to bool) \end{aligned}$$

Figure 6.1: Test-case structure using operation arguments from an external generator

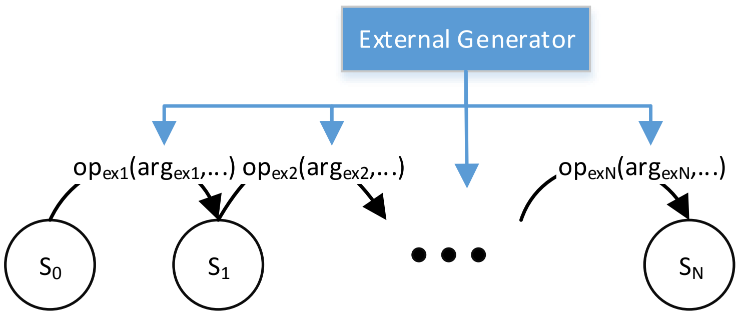

Figure 6.2: Test case (sequence of operations) utilizing externally generated arguments.

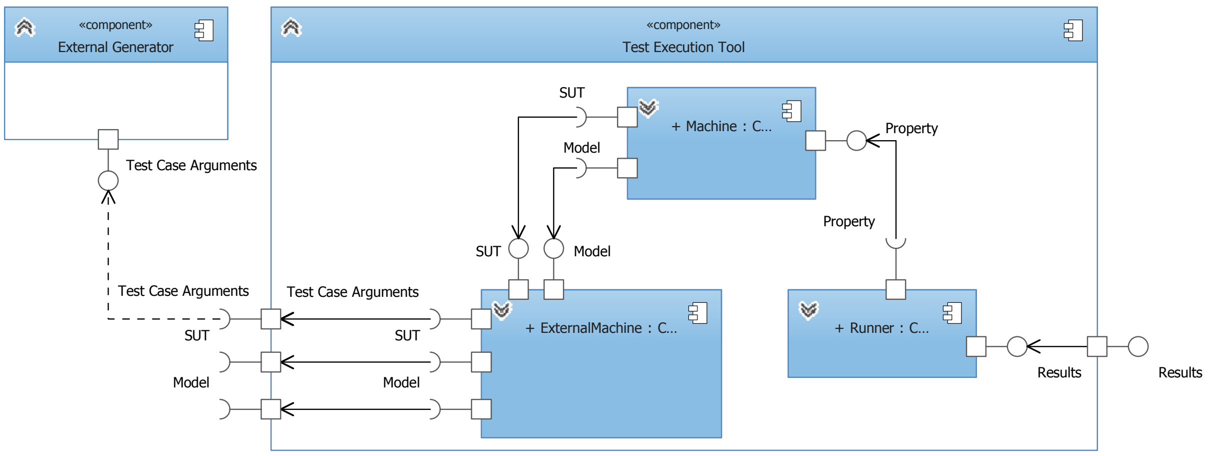

Figure 6.3: Component diagram showing how the FsCheck test-case generation process is extended to support external generators.

already provided by FsCheck and most parts are hidden from the user. The *ExternalMachine* is responsible for transforming the external test arguments into operations and forwarding the SUT and model to the machine. An external generator has to be linked to the *ExternalMachine* to provide the test-case arguments. There are two types of arguments to create test sequences, *SetupArg* and *OperationArg*. They will be explained later in this section. Usually, the external generator will also require information from the SUT or the model. This is not shown in the diagram since it is not necessarily required.

Note that the *ExternalMachine* is abstract, this means the user will have to implement how the arguments are used to generate test sequences himself. In the following, the implementation of the *ExternalMachine* and how the arguments can be used is explained. The implementation is shown in Listing 6.1.

The first notable extensions are the two new generic types: *SetupArg* and *OperationArg*. Those are the arguments that contain the information to generate the initial state and the operations in the *Setup* and *Next* functions. The machine contains a queue to store the data needed for test-case generation. Each element in the queue represents a single test case. This element is an object of the *OperationArguments* class. A *MachineRunArguments* object contains a *SetupArgument* and a queue of *OperationArguments*. During a regular *FsCheck* test case, setup is called in the beginning to create the initial state of the model and then operations are generated by calling *Next* every time. It is possible to supply external information to the *Setup* and *Next* functions through the *MachineRunArguments* of the *ExternalMachine*. The operations can be supplied directly or they can be created from the arguments in the generation process. By filling the outer queue with *OperationArguments* the length of the test case is defined and not variable, as it is usually the case in *FsCheck*. How this is accomplished

```
1  public abstract class ExternalMachine<Actual, Model, SetupArg, OperationArg>
2    : Machine<Actual, Model> {
3      public Queue<MachineRunArguments<SetupArg, OperationArg>> TestCases
4        { get; protected set; }
5      MachineRunArguments<SetupArg, OperationArg> currentTest;
6
7      public ExternalMachine(Queue<MachineRunArguments<SetupArg,
8        OperationArg>> testcases) : base(int.MaxValue) {
9        TestCases = testcases;
10       }
11
12     public sealed override Arbitrary<Setup<Actual, Model>> Setup {
13       get {
14         if (currentTest == null || (TestCases.Count != 0 && currentTest.Count == 0))
15           currentTest = TestCases.Dequeue();
16         return SetupArbitrary(currentTest.SetupArgument);
17       }
18     }
19     public abstract Arbitrary<Setup<Actual, Model>> SetupArbitrary(SetupArg arg);
20
21     public sealed override Gen<Operation<Actual, Model>> Next(Model m) {
22       if (currentTest.Count == 0)
23         return Gen.Constant((Operation<Actual, Model>)
24           new StopOperation<Actual, Model>());
25
26       var arg = currentTest.Dequeue();
27       return Next(m, arg);
28     }
29     public abstract Gen<Operation<Actual, Model>> Next(Model m, OperationArg arg);
30   }
```

Listing 6.1: External machine which represents the interface for external generators.

is explained later in this section. Simplified, the interface contains a queue of queues where the outer queue represents the arguments for a test case and the inner queue represents the arguments for an operation. The data in the queues is supplied by external generators and is passed to the PBT tool to generate its sequences. In this thesis, the data supplied by the external tool were generated before executing any test cases. However, it is possible to access the external generator during the test-case execution. This allows us to use feedback from the SUT to generate new test cases.

In Lines 12-18 the *Setup* function is implemented. The function is sealed in order to enforce the use of the new *SetupArbitrary* function in Line 19, which receives a setup argument as input. The *Setup* simply dequeues a test case and assigns it as the current test. The new *SetupArbitrary* function is then called with the setup argument. The user now has to implement the *SetupArbitrary* instead of the *Setup* function. In Lines 21-28 the *Next* function is shown. Similar to *Setup*, this function dequeues an argument which is then passed into the new *Next* function in Line 29. Again, the function is sealed to enforce the use of the new function and the new function has to be supplied by the user in a derived class. In Line 22-24 a stop operation is generated if the case is finished. The stop operation is a functionality that was implemented into FsCheck in order to ease the implementation of this approach. It

```
1  public class BankRegexBasedMachine :
2    ExternalMachine<BankAccount, int, int, string> {
3
4    public BankRegexBasedMachine(Queue<MachineRunArguments<int, string>> tcs) :
5      base(tcs) { }
6    public override Arbitrary<Setup<BankAccount, int>> SetupArbitrary(int ignored) {
7      return Arb.From(new BankAccountSetupArb());
8    }
9    public override Gen<Operation<BankAccount, int>> Next(int m, string opName) {
10     if (opName.Equals("Deposit"))
11       return Arb.Default.PositiveInt().Generator.Select(i =>
12             (Operation<BankAccount, int>)new DepositOperation(i.Get));
13     else if (opName.Equals("Withdraw"))
14       return Arb.Default.PositiveInt().Generator.Select(i =>
15             (Operation<BankAccount, int>)new WithdrawOperation(i.Get));
16     else
17       throw new NotImplementedException("argument cannot be 0");
18   }
19 }
```

Listing 6.2: A regex-based external machine for testing a bank account.

is now part of the FsCheck library. When a stop operation is returned by the *Next* function, FsCheck identifies this as the end of the current test case and stops generating new operations. This can be useful if a model is in a final state or as in our case if the test case needs to be terminated manually. If the functionality of a PBT tool does not support a stop operation, an operation using the null object pattern [77] can be implemented to generate operations that do not change anything on the model or SUT.

6.2 Regex-Based Sequence Integration

To show how an external generator can be included into a PBT tool, a regular expression (regex)-based sequence generator was created and integrated into FsCheck. The generator only serves as an example for demonstration purposes. The generator receives a regex as input to create a sequence of identifiers encoded as a string. These identifiers are then checked to create a specific type of operation. For example, by using the regex $(Operation1) + (Operation2)*$ the generator will create test cases which always start with at least one operation of the first type and then may add operations of the second type.

We will revisit the bank-account example from previous chapters first introduced in Section 3.4.2. In Listing 6.2, an implementation based on the *ExternalMachine* for the bank account is presented.

The *SetupArgument* in Line 6 is of type integer and is ignored. We could use this argument to set the initial value of the bank account. In our example we chose to initialize every account without any money on it to keep the example simple. The initial state of the bank account is generated using the PBT generators. The *OperationArgument* is of type string and contains the name of the operation to be created. In Lines 9-18 the *Next* function is shown. It implements the corresponding abstract function of Listing 6.1. In the previous case (Listing 3.13), one of the two generators for the operations was chosen randomly with the FsCheck function *Gen.OneOf*. The type of operation is decided by the name of the operation (*opName*) which

```
 1  [TestMethod]
 2  public void BankRegexBasedTest() {
 3    var gen = new SequenceGenerator(new string[] { "Deposit", "Withdraw" });
 4    var regex = "Withdraw(Withdraw)+(Withdraw|Deposit)+";
 5    var tests = gen.Generate(regex, 10);
 6
 7    var config = Configuration.VerboseThrowOnFailure;
 8    config.MaxNbOfTest = tests.Count;
 9    new BankRegexBasedMachine(tests).ToProperty().Check(config);
10  }
```

Listing 6.3: A test method showing the regex-based generator.

was generated by the external regex-generator. The value that is withdrawn or deposited is generated as previously, using the tools generators. The property can then be checked as shown in thetest method in Listing 6.3. In order to test the property of an external machine the maximum number of test cases has to be set in the configuration. This is shown in Line 8. A queue of *MachineRunArguments* named *tests*, which contains the sequence of operation names, is passed to the constructor in Line 9.

Via a generator function which contains a regex, the queue of arguments is generated. Each argument contains the needed data to create an operation. It is also possible to supply the whole operation as an argument. Then the queue of arguments is, in fact, a queue of operations. Fare[1], a .NET port for Google's java library Xeger[2] was integrated to generate strings from the supplied regex. Fare is explained in more detail in Section 5.2.2. Listing 6.4 shows a generator function which creates the queue of *MachineRunArguments*. Note that a *MachineRunArgument* is not a test case but rather a structure that contains all the data needed to create the operations for a test case.

For each test case, a string is generated based on the regex with Fare. One generated string is a sequence of operation types which represents a test case. It has to be parsed and stored in a queue in order to be available within our external Machine. The operation names are stripped from this string one after the other and added to the operation argument queue. This queue together with the unused *SetupArgument* forms one test case and is added to the resulting queue in Line 21. Note that a number of test cases is supplied to the function and can be generated to ensure the flexibility of the PBT approach.

A test case executed with the regular FsCheck *Maschine* interface will create sequences which will create both operations approximately as often. Of course, the likelihood of the operations can be changed to favor one operation over the other. However, the flexibility is limited. With a regex, we can focus better on certain aspects of the system by adjusting the regex. A specification that focuses heavily on withdrawing can utilize the following regex `"Withdraw(Withdraw)+(Withdraw|Deposit)+"`. The operation sequence will always start with a *Withdraw* operation followed by 1 to *n* additional *Withdraw* operations. Only after that 1 to *n* operations follow where *Withdraw* and *Deposit* are as likely to be chosen. The minimum length of a sequence is three, while there are at least two *Withdraw* operations at the start. In a specification as simple as a bank-account model, the sequences based on regex might not be very useful since the random generator of a PBT tool will most likely cover enough useful scenarios. With more complex systems this approach can be applied to target certain critical scenarios that are more likely to produce errors. It can also help to target rare operations that

[1]https://github.com/moodmosaic/Fare
[2]https://code.google.com/archive/p/xeger/

```
public Queue<MachineRunArguments<int,string>> Generate(string regex, int numTests) {
  var tests = new Queue<MachineRunArguments<int, string>>();
  var gen = new Xeger(regex);

  for (int i = 0; i < numTests; i++) {
    var s = gen.Generate();
    var operationArgs = new Queue<string>();
    while (s.Length > 0) {
      string foundOperation = null;
      foreach (var op in new string[] { "Deposit","Withdraw"}) {
        if (s.StartsWith(op)) {
          foundOperation = op;
          break;
        }
      }
      if (foundOperation == null)
        throw new Exception("Generated String is corrupt.");
      s = s.Substring(foundOperation.Length);
      operationArgs.Enqueue(foundOperation);
    }
    tests.Enqueue(new MachineRunArguments<int, string>(0, operationArgs));
  }
  return tests;
}
```

Listing 6.4: Generator function that is based on regular expressions.

can only be triggered in certain corner cases.

In Table 6.1 sequences from three different approaches are shown. Regular FsCheck sequences, *Fs*, weighted FsCheck sequences Fs_w where 70 percent withdraw operations and 30 percent deposit operations are generated and sequences from the regex generator, *regex*. In the table, the operations are abbreviated with their first letter. While we only show two sequences per generation approach we can still observe the different tendencies between FsCheck sequences and the ones from the regex generator. As noted before the regex sequences always start with withdraw operations and they follow a clearer pattern than the randomly generated FsCheck sequences.

6.3 MoMuT Integration

In this section, it is shown how to run MoMuT as an external test sequence generator and integrate the generated sequences in a PBT tool to create a specification. How to create MBT tests with MoMuT was shown in Chapter 4. First, the transformation from EFSMs to the object-oriented action system (OOAS) language is broken down. Then, it is explained how to implement observer automata to achieve certain test goals and form a specification for MoMuT. Mutation operators are then applied to create faulty specifications. Finally, the test sequences created by the MoMuT backend are integrated into FsCheck. The method explained here is one of the evaluated methods in the case study for test-case generation. This process is shown in Figure 6.4. The flowchart is a more detailed and specialized version of the flowchart in Figure 5.1 from the previous chapter.

#	*Fs*		Fs_w		*regex*	
1	D:5 $\rightarrow$ 33	W:3 $\rightarrow$ 25	D:2 $\rightarrow$ 78	W:5 $\rightarrow$ 35	W:7 $\rightarrow$ 88	W:5 $\rightarrow$ 17
2	D:6 $\rightarrow$ 39	W:1 $\rightarrow$ 24	D:7 $\rightarrow$ 85	W:1 $\rightarrow$ 34	W:2 $\rightarrow$ 86	W:4 $\rightarrow$ 13
3	D:2 $\rightarrow$ 41	D:1 $\rightarrow$ 25	W:4 $\rightarrow$ 81	D:10 $\rightarrow$ 44	W:7 $\rightarrow$ 79	W:10 $\rightarrow$ 3
4	D:1 $\rightarrow$ 42	D:7 $\rightarrow$ 32	W:3 $\rightarrow$ 78		W:4 $\rightarrow$ 75	W:1 $\rightarrow$ 2
5	D:2 $\rightarrow$ 44	W:2 $\rightarrow$ 30	W:8 $\rightarrow$ 70		W:3 $\rightarrow$ 72	D:7 $\rightarrow$ 9
6	D:10 $\rightarrow$ 54	W:3 $\rightarrow$ 27	W:1 $\rightarrow$ 69		D:6 $\rightarrow$ 78	D:5 $\rightarrow$ 14
7	D:1 $\rightarrow$ 55		D:5 $\rightarrow$ 74		D:2 $\rightarrow$ 80	W:8 $\rightarrow$ 6
8	W:1 $\rightarrow$ 54		W:5 $\rightarrow$ 69		W:1 $\rightarrow$ 79	
9	W:3 $\rightarrow$ 51				W:9 $\rightarrow$ 70	
10	W:3 $\rightarrow$ 48				W:6 $\rightarrow$ 64	
11	D:3 $\rightarrow$ 51					

Table 6.1: Comparison between regular FsCheck and regex generated sequences.

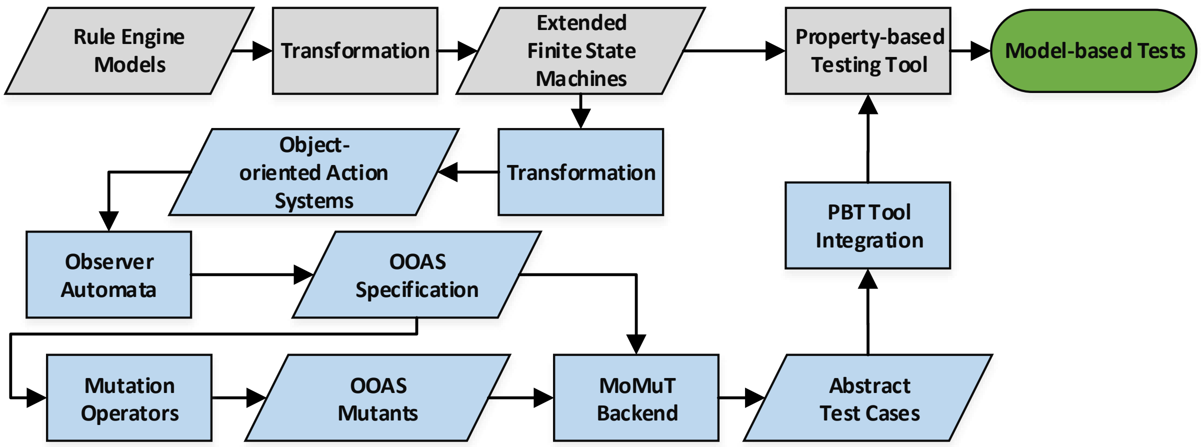

Figure 6.4: Flowchart of the proposed testing approach using MoMuT as an external test-case generator

For this approach, the REMs are translated to EFSMs. The EFSMs are then translated into an OOAS abstract syntax tree (AST). This syntax tree is then modified with observer automata to fulfill certain test goals. The AST represents the original model which is an abstract representation of the input for MoMuT. In order to acquire mutants, the tree is then modified using mutation operators. The original model and the mutants are then translated to a SICStus Prolog program and passed as input for MoMuT's mutation testing. MoMuT then saves the results as abstract test-cases. One abstract test-case is the sequence of operations from the PBT tool's perspective. That means that MoMuT decides on the length of each sequence and supplies the types of operations to the *ExternalMachine* presented in this chapter. The exact data of each operation is still provided by the PBT tool's generators. It is computationally too expensive to model the system at the same level of detail as it was done with the EFSMs. Therefore, mutation-based model testing is only applied to decide on the sequence of operations and the more detailed information, namely the attributes of the tasks, are generated using a PBT tool.

6.3.1 From EFSMs to Object-Oriented Action Systems

Since MoMuT cannot directly work on REMs to generate sequences, they first have to be translated into OOAS to serve as input for MoMuT. REMs and their use in this thesis were explained in Chapter 5. Since the REMs are already parsed into EFSMs and contained in the

$$\begin{aligned}
TL &=_{df} (model, objIdx, objCnt, efsmStates),\ where\\
model &= EnumType(\{efsm.Name \mid efsm \in Module\})\\
objIdx &= IntType(\{-1, ..., n-1\})\\
objCnt &= IntType(\{0, ..., n\})\\
efsmStates &= [EnumType(\{efsm.S\}) \mid efsm \in Module]
\end{aligned}$$

Figure 6.5: Definition of the type list of the abstract syntax tree for object-oriented action systems.

$$\begin{aligned}
VAR =_{df} &\{Variable(currentModel, model, 0)\} \cup\\
&\{Variable(efsm.Name\hat{}"ObjCnt", objCnt, 0) \mid efsm \in Module\} \quad \cup\\
&\{Variable(efsm.Name\hat{}"ObjIdx", ObjIdx, -1) \mid efsm \in Module\} \quad \cup\\
&\{Variable(efsm.Name\hat{}i\hat{}"State", s, efsm.S_0) \mid\\
&\ i \in 0..n \wedge efsm \in Module \wedge s \in efsmStates \wedge s = EnumType(efsm.S)\}
\end{aligned}$$

Figure 6.6: Definition of the variable definition and initialization block.

C# framework developed for this thesis the EFSMs are translated to OOAS rather than the REMs themselves. In order to print the OOAS source code needed as input for MoMuT an abstract syntax tree (AST) of OOAS that emphasizes the EFSM features was integrated. In this section, the translation step from a set of EFSMs to the AST is explained. In the case study, the system consists of multiple modules which again consist of multiple REMs. The AST represents one module, or in other words, a set of EFSMs which got translated from REMs. This is shown in the following definition:

$$Module =_{df} \{BuildEFSM(rem) \mid rem \in REMs\}$$

The AST for OOAS can be defined as a type list TL and the actual action system AS. This results in the following defintion: $OOAS =_{df} (TL \times AS)$. The definition of the type list TL is shown in Figure 6.5.

The type *model* is an enumeration type of the EFSMs in the AST. It indicates which model is currently selected. This corresponds to the *SelectREM* operation of the FsCheck model. The object index *objIdx* and object count *objCnt* keep track of the number of objects that are created for each EFSM and which object is currently selected. The *objIdx* is the currently selected object from the *Select* operation and the *objCnt* limits which objects can be selected. An object that is not yet created cannot be selected. The number of objects that can be created has to be limited in order to avoid state explosion when exploring the model with MoMuT. *efsmStates* is a sequence of enumeration types. For each EFSM such an enumeration type consisting of the states of the respective EFSM is created. They indicate in which state the current EFSM and therefore the model is.

The actual action system *AS* consists of a variable definition and initialization block *VAR* and the set of actions *ACT* which represents the source code for the named actions as well as the action calls in the do-od block. The definition of the AST of the *AS* is the following: $AS =_{df} (VAR \times ACT)$. Listing 6.6 shows the definition of the variable definition and initialization block *VAR*.

The set of variables *VAR* calls the function *Variable* to create the variables. Each variable

$$\begin{aligned} ACT =_{df} \ & \{BuildSelectREM\} \ \cup \\ & \{BuildSelect(efsm) \mid efsm \in Module\} \ \cup \\ & \{BuildDynamic(t, efsm, i) \mid i \in 0..n \land efsm \in Module \land t \in efsm.T\} \end{aligned}$$

Figure 6.7: Definition of the action block.

$$\begin{aligned} BuildSelectREM =_{df} \ & (SelectREM, \quad (m, model), \quad m \neq currentModel, \\ & (currentModel := m;), \quad input) \\ BuildSelect : \ & EFSM \to Action \\ BuildSelect(efsm) =_{df} \ & (''Select''\hat{\ }efsm.Name, \quad (i, efsm.Name\hat{\ }''ObjIdx''), \\ & i \geq 0 \land i \neq efsm.Name\hat{\ }''ObjIdx'' \land i < efsm.Name\hat{\ }''ObjCnt'', \\ & (efsm.Name\hat{\ }''ObjIdx'' := i), \quad input) \end{aligned}$$

Figure 6.8: Build functions for select actions.

consists of an identifier, a variable type, and an initialization value. The $+$ operator symbolizes string concatenation. The *VAR* block consists of a variable that saves which model is currently selected *currentModel*, a variable for each EFSM to count how many objects are created, a variable for each EFSM that saves the currently selected object index of the EFSM and a state variable for each object of each EFSM. Note that in this context each object refers to the amount of possibly created objects which is delimited by setting n. The next part of the AS is the set of actions ACT. It is defined in Listing 6.7

The model has one action to select an EFSM (*SelectREM*), for each EFSM a regular *Select* action will select objects within the EFSM. Finally, we have actions for each transition of all EFSMs in the module. The last set of actions are called *DynamicAction* since they are created dynamically from the REMs. For each type of action, a build function will create the action. In Section 4.2.1 it was explained that an action has a label definition, a parameter list, a guard, a body and a type that marks an action as input, output or internal. All actions that are built in this translation process are marked as input. We will denote an *Action* of the AST as a 5-tuple $a =_{df} (l, p, g, b, t)$, where

l is the label definition,
p is a set of parameters,
g is the enabling function named guard,
b is a set of statements named body,
t is the type of action.

A parameter is a 2-tuple $p =_{df} (id, t)$ where id is an identifier and $t \in TL$ is a type. With the above definition, it is now possible to describe the build functions from the ACT block. The build functions for *SelectREM* and *Select* actions are shown in Listing 6.8.

The two types of actions can be implemented straightforwardly. Their guards ensure that only elements can be selected that are currently not selected and that the objects that get selected already exist. The body of actions only consists of one statement which changes the value of the corresponding variable from the *VAR* block.

Dynamic actions are composed of transitions of the EFSMs. They are the core to changing the model's state. The constrained-based backend does not support complex data types such

$$
\begin{aligned}
BuildDynamic &: Transition \times EFSM \times \mathbb{Z} \rightarrow Action \\
BuildDynamic(t, efsm, n) &=_{df} \begin{cases} BuildCreate(t, efsm, n), & \text{if CanCreate(t)} \\ BuildRegular(t, efsm, n), & \text{otherwise} \end{cases} \\
BuildRegular(t, efsm, n) &= (efsm.Name\hat{}t.Name\hat{}n, \emptyset, \\
&\quad currentModel = efsm.Name \wedge efsm.Name\hat{}''ObjIdx'' = n, \\
&\quad (efsm.Name\hat{}n\hat{}''State'' = efsm.Name\hat{}t.d), \quad input) \\
BuildCreate(t, efsm, n) &= (efsm.Name\hat{}t.Name\hat{}n, \emptyset, currentModel = efsm.Name \quad \wedge \\
&\quad efsm.Name\hat{}''ObjIdx'' = n - 1 \wedge efsm.Name\hat{}''ObjCnt'' = n, \\
&\quad 'efsm.Name\hat{}n\hat{}''State'' = efsm.Name\hat{}t.d, efsm.Name \\
&\quad + ''ObjIdx'' = n,' efsm.Name + ''ObjCnt'' = (n + 1)), \quad input)
\end{aligned}
$$

Figure 6.9: Build functions for dynamic actions.

as lists. MoMuT offers an alternative transformation step to action systems [61]. Since our approach generates the OOAS models from EFSMs we have to unroll list types ourselves. Instead of using an array that contains each object and accesses them via an index, we create separate variables. That has the implication that for each action we need to set the state of the current object explicitly and not via an index. The build function for dynamic actions is shown in Listing 6.9.

The first step in creating a dynamic action is deciding if it creates objects in the SUT or not. Unfortunately, this information was not encoded in the REM and is therefore not part of the syntax. Therefore, determining if a task in the REM creates an object had to be analyzed manually. This information was then stored in a lookup table. The helper function *CanCreate* looks up the value in the table and returns *true* in case the task creates objects.

First, the regular type of dynamic actions is explained. The action name is a composition of the identifiers of the EFSM, the transition and an integer number referring to the object. The set of parameters is empty. The guard of the action is only enabled if the corresponding EFSM and the corresponding object is selected. The action simply changes the state of the object.

Actions that create objects are slightly more complicated. The object index is created with an offset of one since the action itself will create the object which means that initially the index is not set (-1). Furthermore, the object count is restricted. Only if the last created object is selected it is allowed to create a new object. This restriction was implemented since lists are not permitted and in order to keep the model simple. By only allowing the creation of elements if the last object is selected it is precisely defined which index the new object will have. In addition to setting the object state, the index of the currently selected object and the count of objects for the EFSM have to be increased.

It is possible to combine the dynamic actions associated with one transition. Currently, each object has a separate action in order to define the AST concisely. However, it is possible to create another *DiscreteActionBody* that has a parallel composition to distinguish the cases for each object.

In Figure 6.10 state machines of ICM objects and *Select* actions are shown. This is a simplified representation of the ICM module. Note that the *Select* and *IncidentCreateTask* actions are actually valid from any state but were only shown once as they would otherwise obscure the figure.

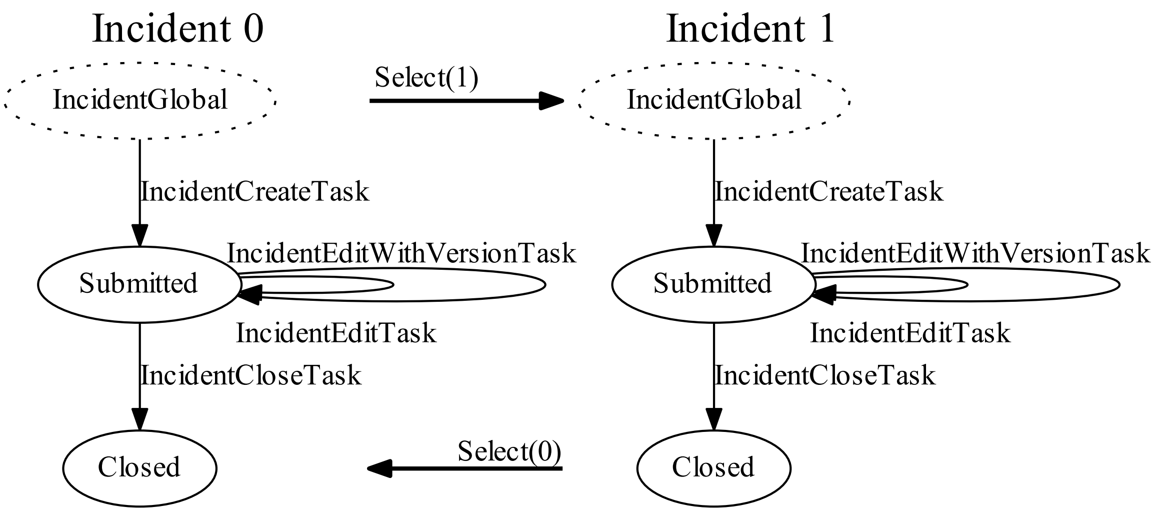

Figure 6.10: ICM module consisting of 2 ICM objects and Select actions.

A simplified OOAS model for the ICM module that is generated by the AST and combines the dynamic actions is shown in Listing 6.5.

Lines 1-5 contain the type list *TL*. Since only one REM is needed to represent the whole ICM module only one state type needs to be created. Following the *TL* the actual action system *AS* is represented in Lines 6-43. In Lines 8-13 the variable block *VAR* of the *AS* is shown. The *currentModel* is only used in *SelectREM* which does not exist in a module with only one REM, therefore it is only needed if multiple REMs exist. The parameter n is set to two, only the allowing creation of two objects per REM. This keeps the example concise. Inside the action block, one action that creates objects and one regular action is shown. The actions for one transition are merged together by using nested require statements. The *Create* transition is shown in Lines 15-22. The source code for the second object is omitted and marked by the dots in Line 22. Also, the source code for the other transitions is omitted. Lines 33-36 show the *Select* action which selects the objects in a REM. The *SelectREM* action is omitted since only one REM is used in the example. The do-od block is shown in Lines 38-42. It contains the calls to the named actions with the correct parameters. Note that an action $a \in ACT$ from the AST is responsible for creating both source code blocks, the named action in the *NamedActionList* and the action call in the do-od block *ActionCallBlock*.

6.3.2 Test Goals via Observer Automata

The presented action system from the previous section has only input actions. No mutation will ever propagate an invalid output since there is no output except the quiescent transitions. This means that it is not possible to perform a useful ioco-check. Adding simple output actions that report the current state would yield us too short test cases as they would be possible in each state. In order to generate longer sequences and achieve test goals we integrated observer automata into the OOAS model. Combining observer automata with mutation testing allows us to generate test cases that can find a difference in the coverage of a model and a mutant. We use the accepting state of the observer as observable output. When non-conformance is found, then we know that the coverage is different, because either the observer of the model or of the mutant are in the accepting state, but not both.

```
types
  Model = {Incident};
  ObjIdx = int [-1..1];
  ObjCnt = int [0..2];
  IncidentState = {IncidentGlobal, Submitted, Closed};
  System = autocons system
  |[
  var
    currentModel : Model = Incident;
    IncidentObjCnt : ObjCnt = 0;
    IncidentObjIdx : ObjIdx = -1;
    Incident0State : IncidentState = IncidentGlobal;
    Incident1State : IncidentState = IncidentGlobal;
  actions
    ctr IncidentCreateSubmitted =
      requires currentModel = Incident and IncidentObjCnt = 0):
        requires IncidentObjIdx = -1 :
          IncidentObjIdx := IncidentObjCnt;
          IncidentObjCnt := IncidentObjCnt + 1;
          Incident0State := Submitted
        end[]
        ...
    end;
    ctr IncidentEditSubmitted =
      requires currentModel = Incident :
        requires IncidentObjIdx = 0 :
          requires Incident0State = Submitted :
            Incident0State := Submitted
          end
        end[]
        ...
    ...
    ctr SelectIncident(i : ObjIdx) =
      requires (i < IncidentObjCnt and i <> IncidentObjIdx and i >= 0) :
        IndexIncident := i
    end
  do
    IncidentCreateSubmitted() []
    IncidentEditSubmitted() []
    ...
    var A : ObjIdx : SelectIncident(A)
  od
  ]|
system
  System
```

Listing 6.5: OOAS for the incident manager module.

Observers can be used to specify coverage criteria for offline test-case generation for systems that have EFSMs to describe the system specifications [49]. Coverage criteria usually consist of a list of items that need to be covered. This can be locations (states) or edges (transi-

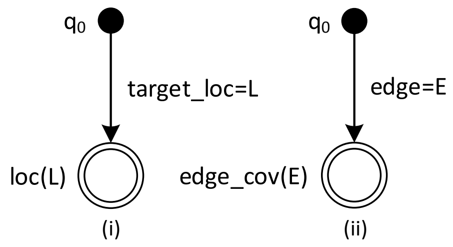

Figure 6.11: Observer automata monitoring location and edge coverage.

tions) in an EFSM. The observer observes how an EFSM traverses its model and saves chosen aspects of the execution. Observers were introduced by Blom et al. [24] and the following definition was introduced in their work. Formally an observer *obs* is defined as a 4-tuple (Q, q_0, Q_f, B) where

> Q is a finite set of observer locations,
> q_o is the initial observer location,
> $Q_f \subseteq Q$ is a set of accepting observer locations, whose names are the corresponding coverage items,
> B is a set of edges, each of the form $q \xrightarrow{b} q'$, where $q, q' \in Q$.

b is a predicate that can depend on the input event received by the SUT, the mapping from state variables of an EFSM to their values after performing the current computation step, and the transition in the EFSM that is executed in response to the current input event.

In the case study, two observer automata were implemented. One for location (state) coverage and another one for edge (transition) coverage. In the context of observer automata and its graph-based terminology we use the locations and edges to refer to the EFSM's states and transitions. The observer automata are shown in Figure 6.11.

The initial location q_0 has an edge to itself with $b = true$. The $\bullet$ symbol represents the initial location with such a self-loop. Furthermore, each $q_f \in Q_f$ has an edge to itself with $b = true$. This is represented by the $\circledcirc$ symbol. The self-loops are added to allow the observer to non-deterministically start monitoring the EFSM and to remain in any of the accepting observer locations. The state-coverage observer (i) has a parameterized accepting location $loc(L)$, where L is a parameter that ranges over all locations in the EFSM. The location $loc(L)$ is entered when the EFSM enters all locations $l \in L$. The transition-coverage observer (ii) works in a similar way where E is the set of all edges from the EFSM. The accepting state is entered as soon as all edges of the EFSM are covered.

The language of observer automata is rich and contains more syntax than what is needed in this thesis. Predicates that make use of variable definitions and variable usages are also included. However, the model exploration of model-based mutation testing is rather expensive. Therefore, only simple observer automata have been implemented in combination with MoMuT. Note that the features of the observer automata language can be exploited more exhaustively with external test-sequence generators that are able to explore their models faster.

The observers are added to the OOAS model by extending the models described in the previous section. For each parameter of the coverage items, a boolean variable has to be added to the *VAR* block. For the transition observer a variable is created for each transition and added to the var block:

$$VAR \cup \{Variable(e_i, boolean, true) \mid e_i \in E\}.$$

Furthermore, those variables have to be set if an edge has been covered. The AST is

```
types
  ...
  System = autocons system
  |[
  var
    currentModel : Model = Incident;
    IncidentCreateSubmittedCrossed : bool = false;
    IncidentEditSubmittedCrossed : bool = false;
    ...
  actions
    ctr IncidentCreateSubmitted =
      requires currentModel = Incident and IncidentObjCnt = 0):
        requires IncidentObjIdx = -1 :
          ...
          IncidentCreateSubmitted := true
        end[]
        ...
    end;
    ...
    obs TransitionCoverageGoalReached =
      requires (IncidentCreateSubmittedCrossed = true and
        IncidentEditSubmittedCrossed and ...) :
          skip;
    end
  do
    IncidentCreateSubmitted() []
    ...
    TransitionCoverageGoalReached()
  od
  ]|
system
  System
```

Listing 6.6: Extended OOAS with a transition-coverage observer.

traversed and whenever an edge is crossed a statement setting the corresponding variable to true is added to the AST. Last, the accepting location *edge_cov*(E) has to be added to the model. The accepting location is modeled as an output action and added to the *ACT* block:

$$ACT \cup \{('TransitionCoverageGoalReached', \emptyset, '\bigwedge_{e_i \in E} e_i = true', ('skip;'), 'output')\}.$$

If all edges have been visited (all variables have been set to true) the action is enabled. The action does not contain any statements and solely skips its execution. Listing 6.6 shows the

The state coverage observer works in a similar manner. For each state, a variable is created and an output action is added to the AST that is enabled when all states are visited.

6.3.3 Test-Sequence Integration

In the previous section, we added output interactions to the OOAS model. This makes it eligible for test-case generation. MoMuT supports random test-case generation. However,

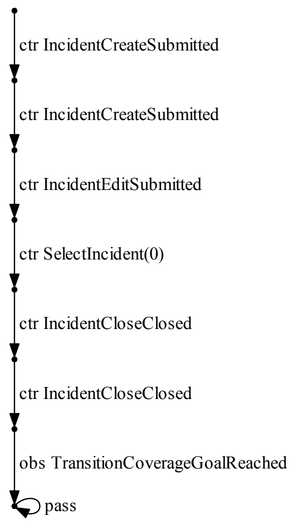
ctr IncidentCreateSubmitted
ctr IncidentCreateSubmitted
ctr IncidentEditSubmitted
ctr SelectIncident(0)
ctr IncidentCloseClosed
ctr IncidentCloseClosed
obs TransitionCoverageGoalReached
pass

Figure 6.12: Abstract test-case using an transition observer for the incident manager.

this was only done to confirm that the generated model can produce the same test sequences as the model generated by FsCheck. Since FsCheck already provides a random test-case generation, generating random test-cases with MoMuT would only compare the two random algorithms of the tools on the models.

The second test-case generation strategy implements a conformance check between a specification and a mutated version. The model from the last section serves as the specification for the MoMuT back end. Since the mutation operators for the MoMuT version we used only works on the level of the UML diagrams, mutation operators are specified by ourselves. We used the symbolic back-end of MoMuT directly with one custom mutation operator. The back-ends were explained in more detail in Chapter 4. While there exists a variety of possibilities for mutation operators, we only tried two different mutation operators. One that modifies the destination state and one that modifies the guard or in other words the starting state. Since both mutation operators seemed to generate similar test-cases only the first mutation operator was chosen for the case study. The mutator changes the destination state of a transition. In Chapter 5 it was defined that a transition t of the EFSM has a destination state $d \in S$. The mutation operator exchanges d by the mutated destination state d' where $d' \in S \wedge d' \neq d$. One transition of the EFSM is randomly mutated using this mutation operator.

When MoMuT is executed using the specification and several mutants, MoMuT produces abstract test-cases in case an error is propagated to an unspecified output action. A generated abstract test-case in the Aldebaran format is shown

In Figure 6.12 we show an abstract test-case as a LTS. We used a transition observer and mutated the destination state of the action *IncidentEditSubmitted* from *Created* instead of *Closed*. The abstract test-case contains all possible transitions of the model, due to the added observable which only triggers when all transitions have been crossed. With the sequence of controllables in the test case, the mutated version will not be able to execute the action *IncidentEditWithVersionSubmitted* as it is not allowed in the *Closed* state. We can never observe that all transitions are crossed. This different behavior is identified by MoMuT and, therefore, the abstract test-case is generated.

Each controllable transition of the test case corresponds to a task of the *REM*. In order to implement the sequence in FsCheck, we make use of the *ExternalMachine* described earlier in this chapter. The implementation of the transformation process of the Aldebaran files to the

```
public override Gen<Operation<IModule, ModelEFSM>> Generator(Model m, string name) {
  var t = m.ActiveRuleEngineModel.Transitions[name];
  var attGen = Generators.GenerateData(t.RequiredAttributes.Values.ToArray());

  return attGen.Select(data => (Operation<IModule, ModelEFSM>)
    new DynamicOperation(t.Name, data, NextState));
}
```

Listing 6.7: Generator for creating dynamic operations based on external machine arguments

external machine is not explained in detail since it is rather straightforward. The explanation is kept short. The important techniques for implementation were already explained in the previous chapters.

The controllables in the sequence get parsed to one of the following three transition types: Dynamic, Select and SelectREM. They can be parsed distinctively with their corresponding attributes. The transition types are the operation arguments for the machine. Each of them has a generator that will generate the corresponding operation. The generator for dynamic operations is shown in Listing 6.7.

The needed arguments for the operation are the model and the name of the transition. The model is created by parsing the REMs to EFSMs and using them in FsCheck. The name of the transition is parsed from the controllables of the abstract test-case. The generators for the *Select* and *SelectREM* generators is implemented similarly. *Select* receives an integer attribute to create the operation while *SelectREM* receives a name to create its operation. With an *ExternalMachine* using the data from Aldebaran files we successfully integrated test sequences, which are created by MoMuT, into FsCheck.

7 Case Study: Testing a Web-Service Application

The approach in this thesis was developed for testing a web-service application. The application was provided by the industrial partner of the TRUCONF project, AVL.[1] The tested system is called AVL Testfactory Management Suite (TFMS) and is used in the automotive industry.[2]

The tool integrates and interacts with test-automation systems and other business systems such as project-management systems, unit-under-test databases. The tool has three main areas, resource management, data & information management, and workflow management. These areas consist of many modules. Each of them is responsible for different functionality such as managing test equipment, reporting issues, planning and preparation of test plans and others.

Three modules were analyzed in depth in the study. The Incident Manager (ICM), the Test-Equipment Manager (TEM) and the Unit-Under-Test Manager (UUT). Other modules that were analyzed include the Test-Order Manager (TOM) and were used to further develop the approach.

The ICM was chosen since it has the smallest EFSM of the tool and is, therefore, the easiest to start implementing and presenting new approaches. The ICM served as an example in the previous chapters. As the ICM is small, it was not further analyzed in the case study.

The TEM and the UUT are one of the bigger sized modules in terms of the EFSMs states, tasks, and transitions. They were selected in order to validate the developed testing approach and to show the viability and limits of the approach for more complex systems. The results of the TEM and the UUT case study are presented in this chapter.

The evaluations in this study were performed with a Lenovo T450s notebook running Windows 8.1 64-bit version. The CPU has 4 logical Intel i5 cores with 2.2GHz and 8 GB RAM. The test cases for the SUT were run on a virtual machine with Windows Server 2008 R2 64-bit version. We assigned 4GB RAM assigned to the machine.

7.1 Test-Equipment Manager

The REMs analyzed in this module are shown in Figure 7.1. The tests were focused on dynamometers (dynos). The approach can be easily adapted for other equipment types as well. The REMs are constructed similarly. We skip the required test data associated with transitions. It can be seen that the REM for the test equipment has a number of tasks to manage/edit the test equipment and also that these tasks lead to different states representing the availability of the test equipment. Note that the models for this case study were more complex than the figure might suggest because we had to consider several instances of the equipment. Hence, we also added operations to switch between instances of this REM. Each test equipment has a test-equipment type. The model of a test-equipment type is similar to the model of a test equipment. The main function of the TEM module is the administration of equipment. Equipment is grouped into base equipment types such as dynamometers, sensors, testbeds, measurement devices, input/output modules and many others. Equipment can be created, configured, edited, calibrated and maintained in this module. In the TEM, we analyzed two REMs: Test equipment (TE) and test-equipment type (TET). TE and TET will refer to the REMs while the written-out version refers to the objects.

[1]https://www.avl.com
[2]https://www.avl.com/tfms

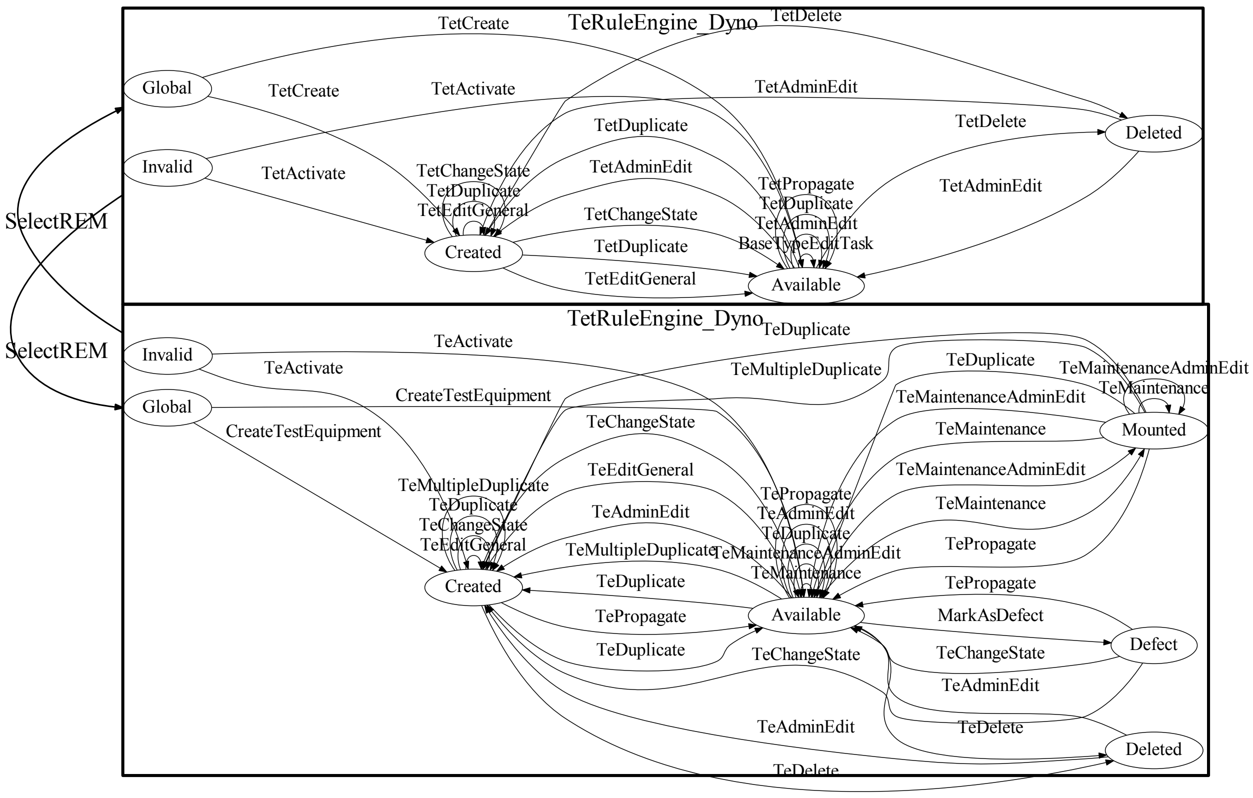

Figure 7.1: EFSM for the rule-engine models of the Test-Equipment Manager module.

Model	States	Tasks	Transitions	Attributes
TestEquipmentType	4 (5)	8 (10)	16 (21)	35 (43)
TestEquipment	5 (7)	7 (13)	13 (39)	18 (23)
TestEquipmentManager	9 (12)	15 (23)	29 (60)	53 (66)

Table 7.1: Number of states, tasks and transitions in REMs of the TEM module

Table 7.1 shows how many states, tasks, transitions, and attributes of the two REMs were tested. The model was only partly tested since some transitions and states were not fully supported or not implemented because they were special cases. The numbers in the parentheses represent the total numbers including these untested items. *TestEquipmentManager* is a model that represents both REMs, *TestEquipmentType* and *TestEquipment*. All edges in the previous graph that share the same label are transitions of the same task. The number of transitions can be calculated with the following summation formula that uses the REM syntax from Section 5.1:

$$transitions = \sum_{s \in AllStates} \sum_{t \in s.possibleTasks} |t.possibleNextStates|$$

Each state has a set of possible next tasks where each of those tasks has a set of possible next states. We count the number of elements of each of sets of possible next states. This is denoted in the formula by summing up the cardinality of the finite *possibleNextStates* sets.

7.1.1 Found Issues

It is important to note that the case study was performed with test REMs, where less test effort is spent than on productive REMs. Productive REMs, which are shipped to the customers, are different ones and were not in the scope of this study. This section shows issues found in the test REMs and shows the ability of the approach to detect these kinds of issues. If the productive REMs contain similar types of errors they should be possible to find. The issues in this section were already presented by Aichernig and Schumi [8]. All issues were found with our approach where we integrated MoMuT into FsCheck but also with the regular FsCheck approach.

The following two issues could be found with strings by utilizing our string generators, which support the generation of strings with regular expressions.

1. Inconsistency regarding the use of tabs in names could be found. It was never planned that the object names should support tabs. On some occasions these characters were replaced with blanks, but not consistently. Blanks were still saved in the DB and only replaced, when they were sent to the GUI. Therefore, two entries could be created that were indistinguishable, because both a name containing a tab and a blank were presented in the same way by the SUT.
2. Another problem we could find was that the regular expressions for several names in our REMs were insufficient. We assume that these regular expressions were designed to prevent certain special characters and no blanks should be allowed at the end and the beginning of the name string. However, the regular expressions were written so that they allowed all non-white space characters at the beginning and the end of the string, even characters that are not allowed in the middle of the string. We could observe this issue when we tested the copy functionality, which duplicates an object and appends an underline and a number to its name. When certain white-space characters were at the end of the string, then the name was not valid anymore, after a copy operation. This was because the special character moved from the end to the middle of the string, where they were not allowed due to the regular expressions.

Additional issues could be found concerning misconfigurations in the REMs and unsupported functionality of the provided test framework.

3. An issue was found with required attributes. In a particular task an attribute was required, but it could not be edited as it was not enabled for this task. Therefore, it was not possible to complete this task, except the user returned to a previous task and edited the attribute there.
4. We found a task that was not supported by the test framework. The task could be triggered with the test framework but resulted in an exception. In the GUI the task could be executed normally. Hence, we found a task that was not completely implemented in the test framework and could not be tested automatically, because without support of the test framework only a manual test via the GUI was possible.

7.1.2 Experiments

Experiment 1: To evaluate our approach we integrated MoMuT into FsCheck by using our interface for external test-case generators described in the previous chapter. The first experiment reports the exploration times of the MoMuT test-case generation process for different depths of the ioco check. In order to do that we will first take a look at the test-case generation-process of MoMuT.

In order to generate tests, certain parameters have to be set. The settings are shown in Table 7.2. n is the maximum number of objects for each REM. m is the number of mutants that are created. *ioco* is the maximum depth of the ioco check. ref is the maximum depth of the refinement check which is explained in more detail in Section 4.3.2. If no error is found

Name	Value
n	5
m	10
ioco	10
ref	20

Table 7.2: Configuration for MoMuT test-case generation

Model	Strategy	d_2		d_4		d_6		d_8		d_{10}	
		time	tc	time	tc	time	tc	time	tc	time	tc
TE	States	3.60s	0	25.76s	4	50.47s	10	5.02m	9	1.56m	10
TE	Tasks	9.01s	0	3.06m	0	54.06m	2	13.45h*	-	213.63h*	-
TET	States	4.34s	0	38.10s	9	1.65m	8	11.54m	9	53.51m	9
TET	Tasks	6.41s	0	2.58m	0	31.10m	0	8.56h*	-	140.63h*	-
TEM	States	5.41s	0	1.13m	0	10.19m	0	7.00h	2	41.75h*	-
TEM	Tasks	26.78	0	8.52m	0	1.70h*	-	23.01h*	-	312.18h*	-

Table 7.3: Exploration times of TEM REMs using MoMuT with observer automata.

within the maximum depths, no abstract test case will be generated. It was observed that the runtime of the test generation-process is at least exponential to the ioco depth. The depth was increased step by step until the process took too long or the depth of 10 was reached. Table 7.3 shows the exploration times for different models and observer strategies.

The models TE and TET refer to the REMs for test equipment and test-equipment types. The TEM model combines the two REMs into a more complex model. We alter the models for MoMuT with observers to reach coverage criteria. The column *Strategy* identifies the type of generation-process via the coverage goal of the observer.

d_n stands for the ioco depth which is set to a number n. For example, d_{10} shows the time it took for the exploration if *ioco* is set to ten. For some strategies MoMuT could not generate any test-cases given a certain ioco depth. Since the exploration times for higher depth values were too long, it was not possible to generate test cases for all strategies. If the depth could not be reached an estimation is given and the time value is marked with an asterisk ($*$). The estimation is calculated using exponential regression. Under the assumption that our data points follow an exponential model of form $f(x) = a^{b*x}$, we estimate the parameters, a and b. This means we can estimate the duration d by inserting a certain *ioco* depth into the formula as x. For further information on curve fitting the interested reader may refer to Motulsky and Christopoulos [80].

We can observe that the task strategies need more time exploring the model as the state strategies. A higher depth is needed to find mutants. To reach the observable action in the OOAS longer test sequences are generated in the task strategies since more observer items are present there. This means that MoMuT needs to explore longer. In the *TE* with the *States* strategy the d_{10} run needs less time than the d_8 run as it is able to generate ten out of ten test cases. If a mutant is killed and a test case is generated MoMuT does not need to continue exploring the whole model and will continue finding the next mutant. This results in shorter exploration times, as the exploration is aborted. On the contrary, if a mutant is not killed MoMuT has to explore the model until d is reached. Since the exploration is rather expensive the case study sticks to those two simple strategies as the ioco check can be performed within a certain depth. Using more complex strategies such as a transition-coverage observer or a combination of a state and task-coverage observer would require a higher depth values and, therefore, longer exploration times. As expected, the bigger the model the longer the exploration times and the fewer mutants can be found in time. When both REMs are combined

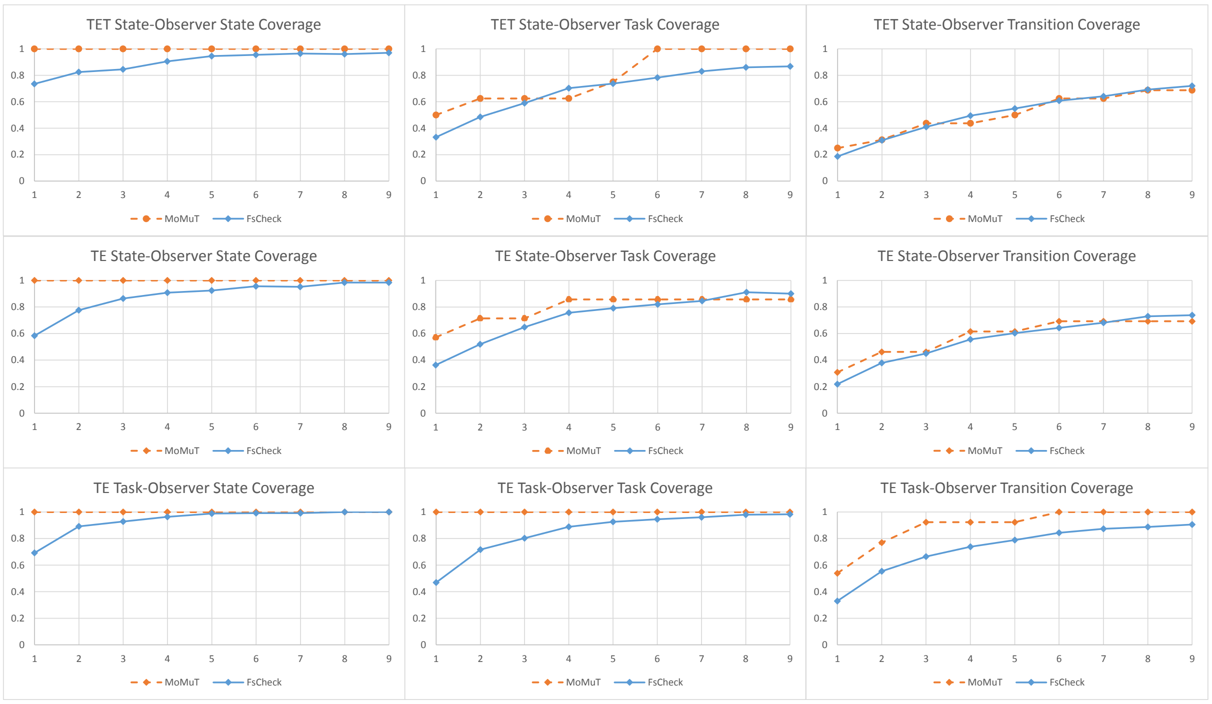

Figure 7.2: Coverage per number of test cases for different models and observer strategies for the TEM module.

exploration times are the longest and also produce the least test cases.

Experiment 2: In this experiment we will compare how adequate our approach and the regular FsCheck approach cover the model. The model coverage is analyzed as follows. The abstract test cases from the MoMuT generation process serve as a reference. For a comparison of the coverage items the amount and length of test cases are controlled. The FsCheck test run is set to the number and length of the MoMuT test-cases. In the following we will show the coverage of the different strategies and models. For a strategy a test suite is generated composed of x test cases of a fixed length. The length is four for the two state-coverage observers and eight for the task-coverage observer. The number of test cases per suite is plotted on the x-axis. On the y-axis the coverage of the model is plotted in percent. Each graph contains two data series. One for the test cases generated with MoMuT and one for the test cases generated with regular FsCheck. The analyzed coverage criteria are state, task and transition coverage. The experiment is repeated 50 times for the FsCheck sequences and the average value is plotted. Since the MoMuT generation used does not involve random generation, it only needs to be performed once. Repeating the experiment multiple times would lead to the same MoMuT sequences.

In Figure 7.2 the different models and observer strategies are shown for the TEM module. For the state observer it can be seen that the state coverage is 100 percent with only one test case for the MoMuT approach. This is expected and a validation that the state observer works as intended. The task-observer strategy shows that state, as well as task coverage, is 100 percent for the MoMuT approach. In our models no unreachable states are included. This means if all tasks are covered all states are covered as well. This is not evident on all models, however, in our examples it is. In all strategies it can be observed that MoMuT covers the model better for smaller test suites. For larger test suites both methods are able to cover most of the model. The MoMuT sequences are also guaranteed to cover certain criteria with only one test case based on the observer type. Since FsCheck relies on random testing no matter how many test cases are generated, it can not be guaranteed that full coverage is achieved with the test suite.

Issue	FsCheck	MoMuT
1	467.4	259
2	12.4	4.4
3	17.6	7.8
4	9	14.2

Table 7.4: Average number of operations needed to find an issue with a given test suite.

Experiment 3: We created two test suites and show how good they perform on finding bugs in the system. We compare with how many operations each test suite is able to find the issues that were listed. The first test suite is plain FsCheck and consists of eight test cases of the length five. The second test suite is created through a mixed approach. We use MoMuT to generate two random tests of the length five and in addition six test cases which were retrieved by trying to kill 50 mutants using a state observer. The test suite consists of 39 operations. A third test suite that uses a task observer would be an interesting addition. Unfortunately, a test suite could not be generated. The model needed to find the issues would contain almost all tasks of the TE, which is infeasible. The test-case generation was aborted after about 30 hours of exploration. All issues were contained in the TE. Therefore, the TET was not part of this experiment.

Table 7.4 contains the number of operations that are needed on average to reproduce the reported issue. The test suite was continuously executed until the issue was found. This means that if the test suite could not find the error with the first execution we continued executing. All experiments were repeated five times. It can be seen that only the first issue was not found within one test suite run. The string generator needs to generate a tab which is one of many possibilities. Therefore, the issue is triggered rarely. For the second issue MoMuT performs way better since there is a state in which only the faulty task is enabled. Every MoMuT test is able to reveal this issue. The third and fourth issue are tasks that are available in multiple states. The task related to Issue 3 is generated more often by MoMuT than the one related to Issue 4. From an abstract perspective both tasks seem similar. They can be executed from a similar number of states and have a similar number of destination states. There is no clear reason as to why MoMuT is faster in finding one issue but not the other. It can be seen that the MoMuT test suite finds issues slightly faster than the FsCheck one. The task-observer test-suite would be able to identify the Issues 3 and 4 even faster since it is guaranteed that the related tasks would be apparent in each test sequence.

7.2 Unit-Under-Test Manager

The UUT is a module that enables test bed oriented administration of units under test. As in the TEM the manager distinguishes between base types, regular types, and the instances of these types. In addition, they have line definitions, however, they are not included in this case study.

The base types consist of chassis, components, engines and others. Again, the tests only focus on one of the base types, engines. The instances can be created, edited, changed and maintained in this manager. In the UUT, we analyzed two REMs: Unit-under-test instance (*UUTInstance*) and unit-under-test type (*UUTType*). *UUTInstance* and *UUTType* will refer to the REMs while the written-out versions refer to the objects.

The modules analyzed in this study are shown in Figure 7.3.

Table 7.5 shows how many states, tasks, transitions and attributes were tested of the two REMs. The model was only partly tested since some transitions and states were not fully supported or not implemented, because they were special cases. The numbers in the parentheses

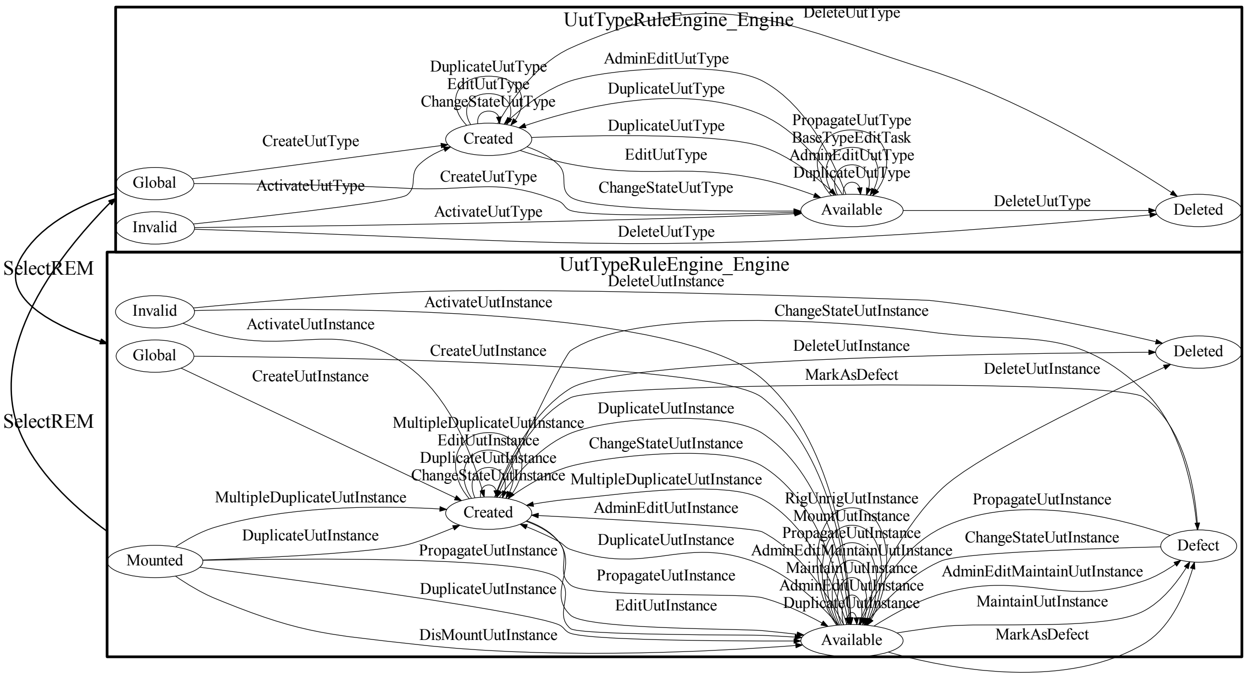

Figure 7.3: EFSM for the rule-engine models of the Unit-Under-Test Manager module.

Model	States	Tasks	Transitions	Attributes
UUTType	3 (4)	6 (8)	12 (16)	25 (45)
UUTInstance	4 (5)	7 (13)	17 (29)	16 (28)
UUT	7 (9)	13 (21)	29 (45)	41 (73)

Table 7.5: Number of states, tasks and transitions in REMs of the UUT module

represent the total numbers including these untested items. *UUT* is a model that represents both REMs, *UUTType* and *UUTInstance*. All edges in the previous graph that share the same label are transitions of the same task.

7.2.1 Found Issues

As described in more detail in the TEM case study, it is important to note that the case study was performed with test REMs, where less test effort is spent than on productive REMs. Productive REMs were not in the scope of this study. As in the TEM module this module had similar issues regarding the regular expressions in name attributes.

The following issues found are in regard to minor misconfiguration of attribute parameters in the REM or to unexpected design choices which can be adjusted in the test framework.

1. An attribute was marked as optional and disabled in a task. The task could not be completed with the attribute missing. By supplying the attribute an error occurred that states that the attribute is not enabled and therefore not allowed. Enabling the attribute in the REM should allow the task to complete properly. This is a minor inconsistency in the rule engine. The task is redundant, since other tasks already implement the behavior.
2. For a UUT a parent type has to be specified via an attribute. UUTs cannot have a base type as their parent, only regular types are allowed. The query to find available parent types, however, returns all types, including base types and types from other sub-types. For example, if the selected element is an engine the parent query may return a chassis as a parent. Narrowing down the query to regular types and the corresponding sub-type

circumvents this behavior. The change is cosmetic and not a failure of the system.

Additional issues could be found regarding state transitions. The REM was not coherent with the SUT in some states. Most of these issues could be fixed by a reconfiguration of the REM. Usually, the implementation uses the REM configuration to perform its transitions. In some special cases the REM configuration is ignored and overwritten hard-coded in the implementation. The found issues are all related to a discrepancy between REM and implementation. The implementation works as expected while the test REM is not implemented cleanly.

3. A certain task is used by administrators to pass changes that were done beforehand by another explicit task. The task is possible to be performed in two different states, however, only one next state is allowed in the REM. After performing the task the UUT remains in its current state. The rule engine is overruled by the implementation which forces the behavior to not change the state.
4. Two other tasks were found that had two possible next states. One of the possible next states could not be set. In one task this resulted in an exception in the test framework, in the other task the task was executed but the state of the UUT was not altered. Again, the rule engine is overruled by the implementation which does not allow a change of state.
5. One task had a single next state. The UUTs changed into a different state and the SUT did not propagate any error. When the REM was adjusted to conform to this behavior and the task was executed the SUT did propagate an error stating that the selected next state is not valid. The error propagates due to a configuration issue of a non-productive REM.

7.2.2 Experiments

The same experiments as for the TEM module were conducted for the UUT module. For an explanation on how the experiments are performed please refer to Section 7.1. Only adjustments and result interpretations will be presented here since the description of the experiments was already given.

Experiment 1: Table 7.6 shows the exploration times for different models and observer strategies. Again, it can be seen that smaller modules need less time to explore and that the tasks strategy is more complex than the states strategy. We can see that the exploration time can be very low if all mutants are found. This can be observed in the states strategy of the *UUTType*. d_8 terminates after only 32.61 seconds even though the estimated time is 5.17 minutes. Since the UUT module is a bit smaller than the TEM module, we were able to retrieve meaningful test cases for the states strategy of the whole model. In conclusion, there were no surprising findings in this experiment. The results confirm the observed behavior of the TEM module.

Experiment 2:

In Figure 7.4 the different models and observer strategies for the UUT module are shown. For the *UUTType* the state observer did not perform very well. This can be seen in the the first row of graphs. The first row also shows a new strategy with the MoMuT kill-check. This will be explained shortly after the other strategies are discussed. The task-observer strategy performs as expected, however, the difference is not as apparent as in the TEM module. The *UUTType* is less complex resulting in FsCheck being able to cover most parts of the model with an adequate amount of operations. The smaller the model is, the less superior the MoMuT approach is in terms of model coverage. The state-observer strategy for the *UUTInstance* performed slightly better than plain FsCheck.

Model	Strategy	d_2 time	d_2 tc	d_4 time	d_4 tc	d_6 time	d_6 tc	d_8 time	d_8 tc	d_{10} time	d_{10} tc
UUTType	States	9.62s	0	43.43s	7	2.91m	6	32.61s	10	9.75m	8
UUTType	Tasks	19.16s	0	5.36m	0	1.07h	0	11.69h*	-	145.1h*	-
UUTInstance	States	9.75s	0	88.29s	9	7.38m	8	18.58m	9	16.87m	8
UUTInstance	Tasks	10.86s	0	5.35m	0	5.16h	0	120.07h*	-	3648h*	-
UUT	States	12.18s	0	1.40m	0	35.97m	1	15.02h	7	122.4h*	-
UUT	Tasks	20.61s	0	5.79m	0		-		-		-

Table 7.6: Exploration times of UUT REMs using MoMuT with observer automata.

Issue	FsCheck	MoMuT
1	10	5.6
2	2.6	2
3	8.6	7.8
4	25.6	26.4
5	23	12.8

Table 7.7: Average number of operations needed to find an issue with a given test suite.

Due to the smaller size of the models, it was possible to create sequences that cover both REMs at the same time. However, the results were not convincing since the MoMuT sequences were performing worse than the FsCheck ones. The reason that MoMuT is not able to cover a big part of the model is that the *UUTType* tends to generate the same sequences over and over again. Since the model is so small there are only a few points in it where discrepancies in model coverage can be observed. Creating the same sequences is definitely not useful. MoMuT supports a kill check. The first row of graphs in Figure 7.4 show that enabling the kill check results in less test-cases but still higher model coverage. When kill check is enabled MoMuT first tries to kill the mutant with the already available test sequences and only if it cannot kill the mutant, it will try to create a new sequence. With this technique no duplicate sequences will be generated and the size of the test suite is kept smaller. The *UUTType* with state observer was reevaluated using the kill check. For 50 mutants a test suite of 6 sequences was created that was able to kill 47 mutants and 3 mutations were identified as ioco equivalent. This strategy for test-suite creation performs significantly better than the plain FsCheck and than MoMuT without a kill check. With only six test cases 100 percent task coverage was achieved.

Experiment 2:

The first issue was contained in the *UUTType*. We used MoMuT with a state-coverage observer to find the first issue. No random test cases were added since they would already kill most of the possible mutation resulting in a test suite of merely two test cases. The generated test suite consists of five sequences with three operations each. The FsCheck test suite contains the same amount of sequences and operations. The other issues were part of the *UUTInstance*. Again, MoMuT with a state-coverage observer was used. However, two random tests of length five were added. Together with nine mutation-based tests, this results in a test suite of 44 test cases. Again, FsCheck uses the same number of test cases and operations.

Table 7.7 shows that the MoMuT generated test suites usually find the issues faster. For the *UUTType* we managed to generate a test suite with a task-coverage observer. The reported issue was found similarly fast as with state-coverage observers, being found after an average of 6.4 operations. For the *UUTInstance* we were not able to generate a suite using the task-coverage observer. We aborted the generation process after about 24 hours. The Issues 3-5

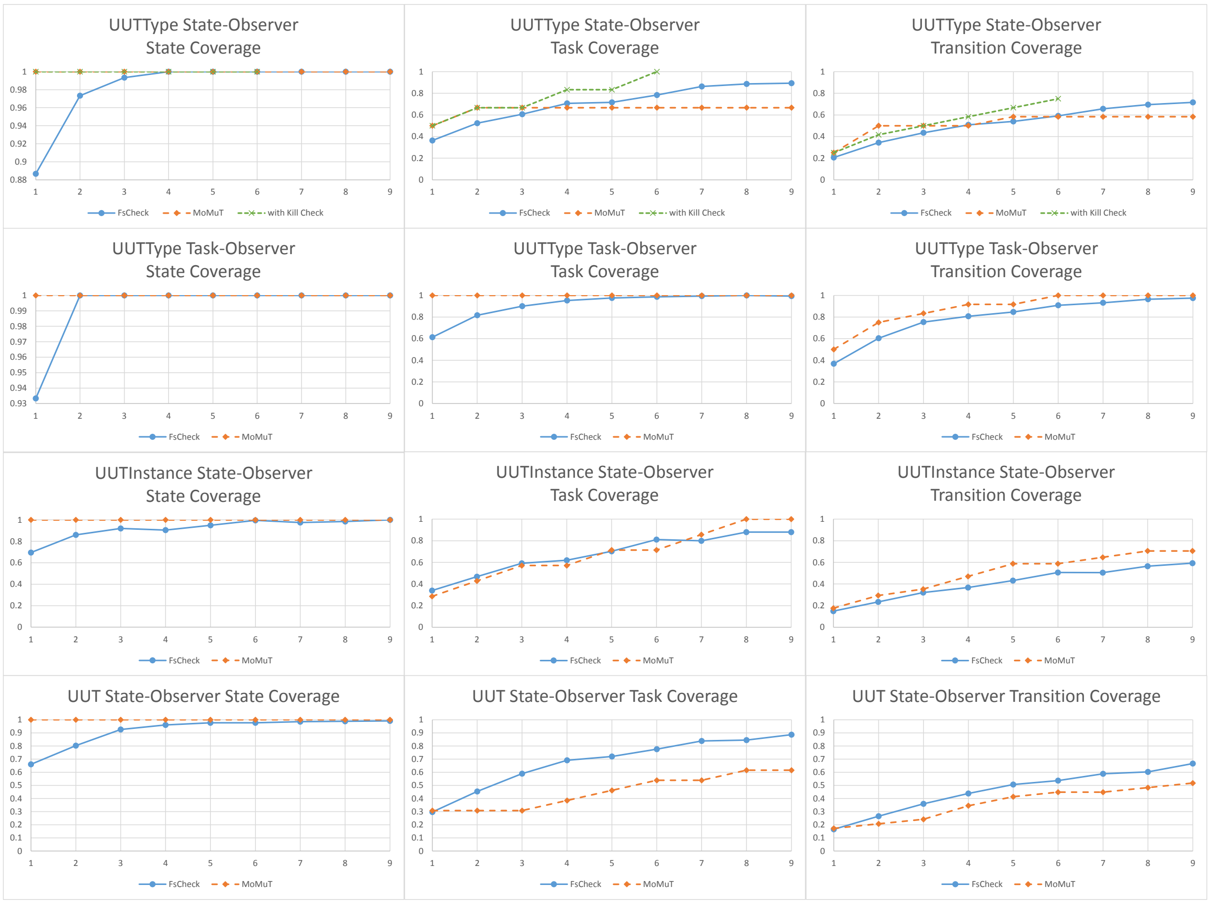

Figure 7.4: Coverage per number of test cases for different models and observer strategies for the UUT module.

would be found with a single test sequence using a task-coverage observer since they are all triggered as soon as a specific task is executed. All of the reported issues of the UUT module are of simple nature and can be found easily. Any test suite that covers all the respective tasks would be able to find the issues. One of the strengths of our approach is that MoMuT finds the issues faster. MoMuT covers a big part of the model with very few test cases. This makes it possible to reveal issues that are straightforward to find with few operations.

7.3 Summary and Outcome

The experiments conducted in this chapter have shown that it can be useful to implement an external test-case generator into a PBT tool. We integrated MoMuT as an external generator, but it is possible to include other test-case generators as well. One of the most notable differences between the MoMuT method and the plain FsCheck method is the difference in computation times. The test-suite generation-time for MoMuT is very high. It can take several minutes to hours to generate useful sequences. With bigger models the problem would become in-feasible to explore. This is a known limitation of the approach. The random strategy of FsCheck can generate sequences of the same length within seconds. Since our MoMuT approach integrates observer automata we can guarantee that a single test case is able to fulfill certain coverage criteria. A test suite that covers all states or all tasks can be guaranteed with this approach. The test suites generated with MoMuT cover more of the model with fewer test cases as compared to plain FsCheck. This advantage becomes neglectable as soon as the amount and length of test cases is increased. It is evident that a smaller test suite will need less execution time. Therefore, it makes sense to optimize for a small test suite if the execution time is expensive. For regression testing the MoMuT sequences can be useful since they are short and cover most parts of the model with little execution time. In addition, it is possible to create test cases that are guaranteed to cover particular parts of the model with observer automata.

The benefits and drawbacks of model-based mutation testing were also discussed by Aichernig et al. [3]. The suggested strategy initializes a test suite with randomly generated test cases. Then, this test suite is run to detect mutants and in case they cannot be killed mutation-based test cases are generated. In this thesis we see that solely relying on mutation-based test cases is insufficient. The models are too big to create a large set of meaningful test cases. Selecting a generation strategy or an appropriate external generator is a non-trivial problem and needs careful analysis. This is coherent with the observations of the authors. The experimental results suggest that mutation-based tests can be more practical for smaller test suites, regression testing or if the execution of test cases is expensive. The regular FsCheck approach, relying on random testing, already provides a very profound basis for creating test sequences. Implementing an external generator is always connected to additional costs and the test-case generation time of MoMuT is very high. If the execution time of running tests is low it is most likely more efficient to generate regular sequences with FsCheck.

8 Conclusion

8.1 Summary

This work presented two testing approaches and combined them. The first is an automatic test-case generation approach for business-rule models and tested a web-service application from AVL called test factory management suite. This initial approach uses a property-based testing (PBT) tool for model-based testing (MBT). The second integrates an external test-case generator into a PBT tool to combine the features of two test-case generation strategies. Both approaches are written in C# and use the PBT tool FsCheck. The second approach uses the model-based mutation testing-tool MoMuT as an external generator.

We have presented a way to derive extended finite-state machines (EFSM) from rule engines and integrate them into a PBT tool FsCheck as input models. It was formally explained how the translation works. MBT with FsCheck was explained with the example of a simple bank account system. Later it was discussed in detail how the EFSM is converted into an object representation that is interpreted by FsCheck as a specification.

Then, FsCheck automatically derives operation sequences from this specification and executes them on the system-under-test (SUT). Operations need some additional data to be executed. This data is expressed by attributes. They contain the information needed for the operation, such as the name of an item that will be created. They usually have a data type and constrains, such as the length of the name. We also generated this attribute data by deriving generators from the rule-engine models. Data types can range from simple to more complex types like strings, objects, references and files. Attributes also support different constraints such as regular expressions for strings, minimum and maximum of numeric values or file extensions.

However, this approach shows some lack of control over the creation of operation sequences. In our case, we wanted to construct test sequences with a different generation strategy in order to focus on other test goals. In order to achieve this flexibility an interface was created to supply a PBT tool with pre-defined operation sequences. We discussed the architecture of our implementation and illustrated the integration method with a bank account example and an external generator based on regular expressions.

In order to test the web-service application from AVL, MoMuT was chosen as an external generator. MoMuT represents its specifications and mutants as object-oriented action systems (OOAS). We have formally shown the translation process from EFSMs to OOAS specifications. Additionally, we introduced how to implement observer automata into the OOAS to fulfill coverage requirements. With the example of a bank account we have shown how MoMuT generates abstract test-cases. The external generator interface presented in this thesis allows us to combine the two testing strategies and embed MoMuT generated sequences into FsCheck. This lets us create a test suite that focuses on fulfilling certain coverage criteria.

This work also covered an application of our new approach which shows the applicability of our test sequence generation strategy in the course of a case study. The case study focuses on comparing different properties between two approaches: The plain FsCheck approach, which was described first in this section and one that integrates MoMuT as an external test-case generator. The experiments conducted in this case study have shown that it can be useful to integrate an external test-case generator into a PBT tool. The conclusions that can be drawn from the case study will be discussed in the following section.

8.2 Discussion

One of the main contributions of this thesis is the implementation of external test-sequence generators into a PBT tool. The developed approach provides an interface extension to FsCheck. The main advantage of our approach is the flexibility that the interface provides. It is obviously still possible to use the tool to generate each operation in the test cases disconnected from other operations. However, with our approach we give the possibility to form new relations in the test-sequence generation-process. While there may exist many more, we have shown two useful applications of our interface. We created a replay mechanism that allows us to save and replay MBT test cases which have more control and is more robust to changes than setting the random seed before executing the test cases again.

We implemented observer automata into the external generator MoMuT to satisfy certain coverage criteria. The main advantage of this approach is that we can guarantee that one test case alone is able to fulfill coverage criteria on the model level. However, the exploration times in our study of MoMuT were at least exponential in relation to the depth of the conformance check. It can take several minutes to hours to generate useful sequences, hence it becomes infeasible for bigger models. This is a known limitation of the approach. In comparison, FsCheck can generate a large number of test cases within one second. It is evident that a smaller test suite will need less execution time. Therefore, it makes sense to optimize for a small test suite if the test-execution time is expensive. The test suites generated with MoMuT cover more of the model with fewer test cases as compared to plain FsCheck. The short MoMuT sequences cover most parts of the model and are therefore well suited for regression testing. However, if the execution on the SUT is cheap it is likely better to implement the plain version instead of paying the expensive costs of test-sequence generation.

There is definitely not one right way to create test sequences. An appropriate approach has to be chosen for each problem. We have provided a solution that is able to combine testing techniques in order to tackle deficits of the individual tools. At the very least we provided a framework in which we can choose the trade-off between computation and execution time and adjust it according to the system and its specific needs.

8.2.1 Concluding Remarks

In conclusion, the two main goals set for this thesis have been reached:

1. Automatically creating models for MBT and including them into the software development process. We derived our PBT models from the system's rule engines and were able to automatize the test generation-process.
2. The integration of external test-case generation-strategies into a PBT tool to gain better control of the means test sequences are generated. This has been achieved by implementing a model-based mutation testing tool MoMuT into FsCheck.

As a result of successfully reaching the set goals, the key parts of this thesis are published as a paper: **Property-Based Testing with External Test-Case Generators** [7]. It was presented at the 13th Workshop on Advances in Model Based Testing (AMOST 2017)[1] in Tokyo, Japan on 17 March 2016.

[1]`http://a-most17.zen-tools.com/`

8.3 Related Work

PBT has gained a lot of attention over the last years. Therefore, it is only natural that a lot of research has been conducted in this field. However, the survey of Dias Neto et al. [39] shows that only a small fraction of research papers have used EFSMs as their model representation. Most of them are based on UML. There exist a variety of approaches that present a combination of MBT and PBT to generate test sequences. The most similar approaches to this thesis are described in this section.

Hughes et al. [53] presented an approach that utilizes QuickCheck that adapts random test-case generation in order to avoid rediscovering the same type of bugs. The approach they presented stores a set of bugs by saving the minimum counterexample to find this type of bug. This information is then analyzed to create sequences that do not contain the already saved sequences. The approach is similar in a way that it utilizes feedback from the test sequences in order to optimize the test-case generation-process. The main difference is that their approach focuses on a specific technique to adapt the test-case generation.

An approach to generate test sequences that cover business rules was presented by Jensen et al. [58]. Business rules are translated into logical formulae and a constraint solver is used to generate the test sequences. This work is similar in a way that test sequences are generated from business rules and that they use a constraint solver for this. In this thesis a constraint solver is used to generate the abstract test-cases provided by FsCheck. While the work is described in a way that it can be interpreted as a variant of MBT, they present their work in a business rule language. Our technique focuses more on translating the business rules into a model and providing more options for the test sequence.

The work of Jensen et al. [59] combines QuickCheck with a fault injection tool. The created testing platform is used to run a quad-copter simulator in order to improve the collision-avoidance mechanism. They inject faults into the simulator and verify the property that the copters do not collide. Similar to the work in this thesis they inject faults in order to acquire test sequences. In the work the model is not created automatically and they focus more on how to use the knowledge of found bugs in order to improve the collision-avoidance mechanism. In comparison to this thesis, they focus more on automating the approach and acquiring test sequences in a different manner, rather than improving the SUT based on the injected faults.

A framework to test web-services with the PBT tool PropEr for Erlang was presented by Lampropoulos and Sagonas [68]. They automatically read the WSDL specification of a web-service to invoke the operations of the web-service with random input. Similar to this work they used data types, but only a few constraints for the data. They implemented automated properties that are not satisfied in case the parsing of the SOAP response encounters an error. In comparison to this thesis they do not use state machines to build their models and other properties have to be implemented by the user.

Francisco et al. [43] presented another similar approach. The framework tests web-services by automatically deriving QuickCheck models from the web-service's WSDL description and OCL semantic constraints. They show how to test stateless and stateful web-services by deriving respective models. In comparison to this thesis their generators do not consider constraints for the data they added the OCL semantic constraints manually. QuickCheck generated the test sequences but unlike in this thesis they did not exploit other test sequence generation strategies.

To the best of our knowledge, we could not find any other work that derives PBT models to automatically test a system and decouples the test-sequence generation-process from the PBT tool giving more control over the generation process.

8.4 Future Work

In the following, further extensions or adaptions of the presented approach are discussed. This should provide information and motivation to possible research paths.

Additional Case Studies: Chapter 7 tries to compare the regular PBT approach with one that combines FsCheck with MoMuT as an external generator. As stated before, the exploration times of MoMuT were quite high. In the scope of the thesis it was not possible to test more parts of the system or to repeat the measurements more often. It would be interesting if the conclusions we drew from the case study also hold for other modules and to increase our confidence by repeating the measurements. It is also interesting how the approach can be embedded into other systems and to find out the limits of our approach. Of course there several other possibilities for further analysis.

Different Models for MoMuT: It could be seen that the time MoMuT needs to explore its models were large. Therefore, we were not able to create complex observer automata to reach different test goals. Using elaborate observer automata could lead to a different type of test sequences. Also the approach would be more practical if the models are explored faster. The models from this thesis were rather complex and more features of MoMuT could be exploited if it is executed on a more trivial model representation. It would be worthwhile to see what is achievable if the exploration times are reduced.

Negative Testing: In this thesis we derived sequences and data from rule-engine models. We created a model to verify that for a given input our model works the same as the system. This was done for sequences that should be allowed and data that is within its limits. This tests the response of the system for normal parameters. The system can also be deliberately fed with wrong data or wrong interactions to determine if the system performs error handling for unexpected input. For example, this can be achieved by inverting the value range of data types. This is a trivial problem for numeric types and could test against integer overflows. However, for more complex types such as strings with regular-expression constraints this is a challenging task worth researching.

External Test-Case Generators: Implementing new external test-case generators is probably the most obvious extension of this research. This should not only be seen as an extension of testing web-services from rule-engine models but can also be applied in entirely different fields. We already presented a generator for replaying test sequences and a regular-expression based generator as examples. It is also possible to implement user heuristics into the generation process. In our case study we have used MoMuT to create test sequences that propagate differences in the coverage of a model and a mutant. However, a generator that solely focuses on coverage criteria should therefore be faster. This would be an interesting addition. With imagination it should be achievable to come up with many variations as to what an external test-case generator might be. There are many promising possibilities yet to be analyzed.

Bibliography

[1] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. *Mutation Analysis*. Tech. rep. Georgia Institue of Technology - Atlanta School Of Information and Computer Science, 1979 (cit. on p. 9).

[2] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. "UML in action: A Two-Layered Interpretation for Testing". In: *ACM SIGSOFT Software Engineering Notes* 36.1 (2011), pp. 1–8. DOI: `10.1145/1921532.1921559` (cit. on p. 30).

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. "Killing strategies for Model-Based Mutation Testing". In: *Software Testing, Verification and Reliability* 25.8 (2015), pp. 716–748. DOI: `10.1002/stvr.1522` (cit. on pp. 27, 75).

[4] Bernhard K. Aichernig and Elisabeth Jöbstl. "Efficient Refinement Checking for Model-Based Mutation Testing". In: *2012 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*. Ed. by Antony Tang and Henry Muccini. IEEE, 2012, pp. 21–30. DOI: `10.1109/QSIC.2012.58` (cit. on p. 33).

[5] Bernhard K. Aichernig, Elisabeth Jöbstl, and Martin Tappler. "Does This Fault Lead to Failure? Combining Refinement and Input-Output Conformance Checking in Fault-Oriented Test-Case Generation". In: *Journal of Logical and Algebraic Methods in Programming* 85.5 (2016), pp. 806–823. DOI: `10.1016/j.jlamp.2016.02.002` (cit. on p. 32).

[6] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. "Model-Based Mutation Testing via Symbolic Refinement Checking". In: *Science of Computer Programming* 97 (2015), pp. 383–404. DOI: `10.1016/j.scico.2014.05.004` (cit. on p. 33).

[7] Bernhard K. Aichernig, Silvio Marcovic, and Richard Schumi. "Property-Based Testing with External Test-Case Generators". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 337–346. DOI: `10.1109/ICSTW.2017.62` (cit. on pp. 3, 77).

[8] Bernhard K. Aichernig and Richard Schumi. "Property-Based Testing of Web Services by Deriving Properties from Business-Rule Models". In: *Software and Systems Modeling* (2017). submitted (cit. on pp. 40–42, 67).

[9] Bernhard K. Aichernig and Richard Schumi. "Property-Based Testing with FsCheck by Deriving Properties from Business Rule Models". In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 219–228. DOI: `10.1109/ICSTW.2016.24` (cit. on pp. 1–3, 35–37).

[10] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. "MobiGUITAR: Automated Model-Based Testing of Mobile Apps". In: *IEEE Software* 32.5 (2015), pp. 53–59. DOI: `10.1109/MS.2014.55` (cit. on p. 14).

[11] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation". In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001. DOI: `10.1016/j.jss.2013.02.061` (cit. on p. 1).

[12] James H. Andrews, Lionel C. Briand, and Yvan Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 402–411. DOI: 10.1145/1062455.1062530 (cit. on p. 13).

[13] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, and Satish Thatte. *Business Process Execution Language for Web Services*. 2003 (cit. on p. 13).

[14] Assaf Arkin. "Business Process Modeling Language". In: *Business Process Modeling Initiative* (2003) (cit. on p. 13).

[15] Ali Arsanjani. "Rule Pattern Language 2001: A Pattern Language for Adaptive Manners and Scalable Business Rule Design and Construction". In: *TOOLS USA 2001: Software Technologies for the Age of the Internet, 39th International Conference & Exhibition, Santa Barbara, CA, USA, July 29 - August 3, 2001*. IEEE Computer Society, 2001, pp. 370–376. DOI: 10.1109/TOOLS.2001.10046 (cit. on p. 12).

[16] Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. "Testing Telecoms Software with QuviQ Quickcheck". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*. Ed. by Marc Feeley and Philip W. Trinder. ACM, 2006, pp. 2–10. DOI: 10.1145/1159789.1159792 (cit. on p. 12).

[17] Ralph-Johan Back and Reino Kurki-Suonio. "Decentralization of Process Nets with Centralized Control". In: *Distributed Computing* 3.2 (1989), pp. 73–87. DOI: 10.1007/BF01558665 (cit. on p. 28).

[18] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013. URL: http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types (cit. on p. 15).

[19] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785. URL: https://doi.org/10.1109/TSE.2014.2372785 (cit. on p. 7).

[20] Antonia Bertolino. "Software Testing Research: Achievements, Challenges, Dreams". In: *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*. Ed. by Lionel C. Briand and Alexander L. Wolf. IEEE Computer Society, 2007, pp. 85–103. DOI: 10.1109/FOSE.2007.25 (cit. on p. 1).

[21] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. "Compositional Testing with IOCO". In: *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*. Ed. by Alexandre Petrenko and Andreas Ulrich. Vol. 2931. Lecture Notes in Computer Science. Springer, 2003, pp. 86–100. DOI: 10.1007/978-3-540-24617-6_7 (cit. on p. 32).

[22] Robert V. Binder, Bruno Legeard, and Anne Kramer. "Model-Based Testing: Where Does It Stand?" In: *Communications of the ACM* 58.2 (2015), pp. 52–56. DOI: 10.1145/2697399 (cit. on p. 7).

[23] Dines Bjørner and Cliff B. Jones, eds. *The Vienna Development Method: The Meta-Language*. Vol. 61. Lecture Notes in Computer Science. Springer, 1978. DOI: 10.1007/3-540-08766-4 (cit. on p. 38).

[24] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. “Specifying and Generating Test Cases Using Observer Automata”. In: *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*. Ed. by Jens Grabowski and Brian Nielsen. Vol. 3395. Lecture Notes in Computer Science. Springer, 2004, pp. 125–139. DOI: 10.1007/978-3-540-31848-4_9 (cit. on p. 61).

[25] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. “An Approach to Object-Orientation in Action Systems”. In: *Mathematics of Program Construction, MPC’98, Marstrand, Sweden, June 15-17, 1998, Proceedings*. Ed. by Johan Jeuring. Vol. 1422. Lecture Notes in Computer Science. Springer, 1998, pp. 68–95. DOI: 10.1007/BFb0054286 (cit. on pp. 28, 29).

[26] Roger T. Burlton. *An Introduction to "The Business Process Manifesto"*. 2012. URL: http://www.brcommunity.com/b672.php (visited on 04/07/2016) (cit. on p. 12).

[27] Mats Carlsson and Per Mildner. “Sicstus Prolog-the First 25 Years”. In: *Theory and Practice of Logic Programming* 12.1-2 (Jan. 2012), pp. 35–66. ISSN: 1471-0684. DOI: 10.1017/S1471068411000482 (cit. on p. 28).

[28] Mats Carlsson, Greger Ottosson, and Björn Carlson. “An Open-Ended Finite Domain Constraint Solver”. In: *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP’97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*. Ed. by Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen. Vol. 1292. Lecture Notes in Computer Science. Springer, 1997, pp. 191–206. DOI: 10.1007/BFb0033845 (cit. on p. 34).

[29] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. “Learning Extended Finite State Machines”. In: *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Vol. 8702. Lecture Notes in Computer Science. Springer, 2014, pp. 250–264. DOI: 10.1007/978-3-319-10431-7_18 (cit. on p. 38).

[30] Wing Kwong Chan, Shing Chi Cheung, and T. H. Tse. “Fault-Based Testing of Database Application Programs with Conceptual Data Model”. In: *Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia*. IEEE Computer Society, 2005, pp. 187–196. DOI: 10.1109/QSIC.2005.27 (cit. on p. 9).

[31] Anis Charfi and Mira Mezini. “Hybrid Web Service Composition: Business Processes Meet Business Rules”. In: *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*. Ed. by Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou. ACM, 2004, pp. 30–38. DOI: 10.1145/1035167.1035173. URL: http://doi.acm.org/10.1145/1035167.1035173 (cit. on p. 13).

[32] Kwang-Ting Cheng and Anjur S. Krishnakumar. “Automatic Functional Test Generation Using the Extended Finite State Machine Model”. In: *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*. Ed. by Alfred E. Dunlop. ACM Press, 1993, pp. 86–91. DOI: 10.1145/157485.164585 (cit. on p. 38).

[33] Kwang-Ting Cheng and Anjur S. Krishnakumar. “Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model”. In: *ACM Transactions on Design Automation of Electrical Systems* 1.1 (1996), pp. 57–79. DOI: 10.1145/225871.225880 (cit. on p. 38).

[34] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* Ed. by Martin Odersky and Philip Wadler. ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266 (cit. on pp. 1, 2, 10).

[35] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. "Model-Based Testing in Practice". In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.* Ed. by Barry W. Boehm, David Garlan, and Jeff Kramer. ACM, 1999, pp. 285–294. URL: http://portal.acm.org/citation.cfm?id=302405.302640 (cit. on p. 7).

[36] Richard A DeMillo. *Mutation Analysis as a Tool for Software Quality Assurance*. Tech. rep. Georgia Institue of Technology - Atlanta School Of Information and Computer Science, 1980 (cit. on p. 9).

[37] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *IEEE Computer* 11.4 (1978), pp. 34–41. DOI: 10.1109/C-M.1978.218136 (cit. on p. 9).

[38] Anna Derezinska and Anna Szustek. "CREAM - A system for Object-Oriented Mutation of C# Programs". In: *Annals Gdansk University of Technology - Faculty of ETI* 5 (2007), pp. 389–406 (cit. on p. 9).

[39] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. "A Survey on Model-Based Testing Approaches: a Systematic Review". In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. WEASELTech '07. Atlanta, Georgia: ACM, 2007, pp. 31–36. DOI: 10.1145/1353673.1353681 (cit. on pp. 2, 7, 78).

[40] Michael Fagan. "Reviews and Inspections". In: *Software Pioneers - Contributions to Software Engineering* (2002), pp. 562–573 (cit. on p. 1).

[41] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. "CADP - A Protocol Validation and Verification Toolbox". In: *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*. Ed. by Rajeev Alur and Thomas A. Henzinger. Vol. 1102. Lecture Notes in Computer Science. Springer, 1996, pp. 437–440. DOI: 10.1007/3-540-61474-5_97 (cit. on p. 28).

[42] George Fink and Karl N. Levitt. "Property-Based Testing of Privileged Programs". In: *10th Annual Computer Security Applications Conference, ACSAC 1994, 5-9 December, 1994 Orlando, FL, USA*. IEEE, 1994, pp. 154–163. DOI: 10.1109/CSAC.1994.367311 (cit. on p. 10).

[43] Miguel A. Francisco, Macias Lopez, Henrique Ferreiro, and Laura M. Castro. "Turning Web Services Descriptions into QuickCheck Models for Automatic Testing". In: *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013*. Ed. by Steve Vinoski and Laura M. Castro. ACM, 2013, pp. 79–86. DOI: 10.1145/2505305.2505306 (cit. on p. 78).

[44] Gordon Fraser, Franz Wotawa, and Paul Ammann. "Testing with Model Checkers: A Survey". In: *Software Testing, Verification and Reliability* 19.3 (2009), pp. 215–261. DOI: 10.1002/stvr.402 (cit. on p. 2).

[45] Lars-Åke Fredlund, Clara Benac Earle, Ángel Herranz-Nieva, and Julio Mariño-Carballo. "Property-Based Testing of JSON Based Web Services". In: *2014 IEEE International Conference on Web Services, ICWS, 2014, Anchorage, AK, USA, June 27 - July 2, 2014*. IEEE Computer Society, 2014, pp. 704–707. DOI: `10.1109/ICWS.2014.110` (cit. on p. 47).

[46] Vignir Gudmundsson, Mikael Lindvall, Luca Aceto, Johann Bergthorsson, and Dharmalingam Ganesan. "Model-Based Testing of Mobile Systems - An Empirical Study on QuizUp Android App". In: *Proceedings First Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@IFM 2016, Reykjavik, Iceland, 4th June 2016*. Ed. by Luca Aceto, Adrian Francalanza, and Anna Ingolfsdottir. Vol. 208. EPTCS. 2016, pp. 16–30. DOI: `10.4204/EPTCS.208.2` (cit. on p. 35).

[47] Peli de Halleux and Nikolai Tillmann. "Parameterized Test Patterns for Effective Testing with Pex". In: *Research in Software Engineering, Microsoft research* 21 (2008) (cit. on p. 16).

[48] Hartmann Software Group. *Business Rule Management System*. 2012. URL: `http://www.hartmannsoftware.com/Blog/Enterprise-Rule-Applications/brms` (visited on 04/07/2016) (cit. on p. 12).

[49] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. "Testing Real-Time Systems Using UPPAAL". In: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Vol. 4949. Lecture Notes in Computer Science. Springer, 2008, pp. 77–117. DOI: `10.1007/978-3-540-78917-8_3` (cit. on pp. 8, 60).

[50] Jay Hilyard and Stephen Teilhet. "Delegates, Events, and Lambda Expressions". In: *C# 3.0 Cookbook*. "O'Reilly Media, Inc.", 2007, pp. 316–365 (cit. on p. 15).

[51] William E. Howden. "Weak Mutation Testing and Completeness of Test Sets". In: *IEEE Transactions on Software Engineering* 8.4 (1982), pp. 371–379. DOI: `10.1109/TSE.1982.235571` (cit. on p. 8).

[52] John Hughes. "QuickCheck Testing for Fun and Profit". In: *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*. Ed. by Michael Hanus. Vol. 4354. Lecture Notes in Computer Science. Springer, 2007, pp. 1–32. DOI: `10.1007/978-3-540-69611-7_1` (cit. on pp. 1, 11, 40).

[53] John Hughes, Ulf Norell, Nicholas Smallbone, and Thomas Arts. "Find More Bugs with QuickCheck!" In: *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*. Ed. by Christof J. Budnik, Gordon Fraser, and Francesca Lonetti. ACM, 2016, pp. 71–77. DOI: `10.1145/2896921.2896928` (cit. on pp. 2, 78).

[54] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. "Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service". In: *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 135–145. DOI: `10.1109/ICST.2016.37` (cit. on p. 2).

[55] Antti Huima. "Implementing Conformiq Qtronic". In: *Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings*. Ed. by Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp. Vol. 4581. Lecture Notes in Computer Science. Springer, 2007, pp. 1–12. DOI: `10.1007/978-3-540-73066-8_1` (cit. on p. 8).

[56] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. “Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria”. In: *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*. Ed. by Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde. IEEE Computer Society / ACM Press, 1994, pp. 191–200 (cit. on p. 13).

[57] Laura Inozemtseva and Reid Holmes. “Coverage Is Not Strongly Correlated with Test Suite Effectiveness”. In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 435–445. DOI: `10.1145/2568225.2568271` (cit. on p. 13).

[58] Simon Holm Jensen, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. “Test Generation from Business Rules”. In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10. DOI: `10.1109/ICST.2015.7102608` (cit. on p. 78).

[59] Simon Holm Jensen, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. “Test Generation from Business Rules”. In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10. DOI: `10.1109/ICST.2015.7102608` (cit. on p. 78).

[60] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: `10.1109/TSE.2010.62` (cit. on pp. 9, 10).

[61] Elisabeth Jöbstl. “Model-Based Mutation Testing with Constraint and SMT Solvers”. PhD thesis. Graz University of Technology - Institute for Software Technology, 2014 (cit. on pp. 28–30, 33, 58).

[62] Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013. URL: `http://www.hackersdelight.org/` (cit. on p. 16).

[63] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. “Are Mutants a Valid Substitute for Real Faults in Software Testing?” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 654–665. DOI: `10.1145/2635868.2635929` (cit. on p. 13).

[64] Mohammad Reza Keyvanpour, Hajar Homayouni, and Hossein Shirazee. “Automatic Software Test Case Generation: An Analytical Classification Framework”. In: *International Journal of Software Engineering and Its Applications* 6.4 (2012), pp. 1–16 (cit. on p. 1).

[65] Sunwoo Kim, John Clark, and John McDermid. *The Rigorous Generation of Java Mutation Operators Using HAZOP*. Tech. rep. The University of York - Department of Computer Science, 1999 (cit. on p. 9).

[66] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. “Mapping UML to Labeled Transition Systems for Test-Case Generation - A Translation via Object-Oriented Action Systems”. In: *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel. Vol. 6286. Lecture Notes in Computer Science. Springer, 2009, pp. 186–207. DOI: `10.1007/978-3-642-17071-3_10` (cit. on p. 10).

[67] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. "MoMuT::UML Model-Based Mutation Testing for UML". In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–8. DOI: 10.1109/ICST.2015.7102627 (cit. on pp. 1, 10, 27).

[68] Leonidas Lampropoulos and Konstantinos Sagonas. "Automatic WSDL-Guided Test Case Generation for PropEr Testing of Web Services". In: *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems, WWV 2012, Stockholm, Sweden, 16th July 2012.* Ed. by Josep Silva and Francesco Tiezzi. Vol. 98. EPTCS. 2012, pp. 3–16. DOI: 10.4204/EPTCS.98.3 (cit. on p. 78).

[69] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. *Uppaal Tron User Manual*. Tech. rep. Aalborg University - Department of Computer Science, 2009 (cit. on p. 8).

[70] Frank Leymann et al. "Web Services Flow Language (WSFL 1.0)". In: (2001) (cit. on p. 13).

[71] Huiqing Li and Simon J. Thompson. "Improved Semantics and Implementation Through Property-Based Testing with Quickcheck". In: *9th International Workshop on Automation of Software Test, AST 2014, Hyderabad, India, May 31 - June 1, 2014*. Ed. by Hong Zhu, Jerry Gao, Saurabh Sinha, and Lu Zhang. ACM, 2014, pp. 50–56. DOI: 10.1145/2593501.2593509 (cit. on p. 11).

[72] J. Jenny Li and W. Eric Wong. "Automatic Test Generation from Communicating Extended Finite State Machine (CEFSM)-Based Models". In: *5th International Symposiun on Object Oriented Real-Time Distributed Computing, ISORC 2002, Washington, DC, USA, April 29 - May 1, 2002*. IEEE Computer Society, 2002, pp. 181–188. DOI: 10.1109/ISORC.2002.1003693. URL: https://doi.org/10.1109/ISORC.2002.1003693 (cit. on p. 38).

[73] Richard Lipton. *Fault Diagnosis of Computer Programs*. Tech. rep. Carnegie Mellon University, 1971 (cit. on p. 8).

[74] Malte Lochau, Sven Peldszus, Matthias Kowal, and Ina Schaefer. "Model-Based Testing". In: *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. Vol. 8483. Lecture Notes in Computer Science. Springer, 2014. DOI: 10.1007/978-3-319-07317-0 (cit. on p. 7).

[75] Nancy A. Lynch and Mark R. Tuttle. *An Introduction to Input/Output Automata*. Tech. rep. Massachusetts Institute of Technology - Laboratory for Computer Science, 1988 (cit. on p. 31).

[76] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation". In: *IEEE Transactions on Software Engineering* 40.1 (2014), pp. 23–42. DOI: 10.1109/TSE.2013.44 (cit. on p. 9).

[77] Robert C. Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design, 3*. Addison-Wesley Professional, 1998 (cit. on p. 52).

[78] Ali Mili and Fairouz Tchier. *Software testing: Concepts and Operations*. John Wiley and Sons, 2015 (cit. on p. 1).

[79] Mehdi MirzaAghaei and Ali Mesbah. "DOM-Based Test Adequacy Criteria for Web Applications". In: *International Symposium on Software Testing and Analysis, ISSTA'14, San Jose, CA, USA - July 21 - 26, 2014*. Ed. by Corina S. Pasareanu and Darko Marinov. ACM, 2014, pp. 71–81. DOI: 10.1145/2610384.2610406 (cit. on p. 14).

[80] Harvey Motulsky and Arthur Christopoulos. *Fitting Models to Biological Data Using Linear and Nonlinear Regression: A Practical Guide to Curve Fitting*. Oxford University Press, 2004 (cit. on p. 68).

[81] Leonardo Mendonca de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 28).

[82] "Mutation-Driven Test Case Generation Using Short-Lived Concurrent Mutants - First Results". In: *CoRR* abs/1601.06974 (2016). Withdrawn. URL: http://arxiv.org/abs/1601.06974 (cit. on p. 28).

[83] Rickard Nilsson. *ScalaCheck: Property-Based Testing for Scala*. 2008. URL: http://scalacheck.org/ (visited on 04/05/2016) (cit. on p. 12).

[84] A. Jefferson Offutt and Jane Huffman Hayes. "A Semantic Model of Program Faults". In: *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996*. Ed. by Steve J. Zeil and Will Tracz. ACM, 1996, pp. 195–200. DOI: 10.1145/229000.226317 (cit. on p. 9).

[85] Lee Pike. "Smartcheck: Automatic and Efficient Counterexample Reduction and Generalization". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 53–64. DOI: 10.1145/2633357.2633365 (cit. on p. 12).

[86] Irith Pomeranz and Sudhakar M. Reddy. "LOCSTEP: A Logic-Simulation-Based Test Generation Procedure". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.5 (1997), pp. 544–554. DOI: 10.1109/43.631218 (cit. on p. 10).

[87] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. "One Evaluation of Model-Based Testing and Its Automation". In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 392–401. DOI: 10.1145/1062455.1062529 (cit. on p. 1).

[88] Jane Radatz, Anne Geraci, and Freny Katki. "IEEE Standard Glossary of Software Engineering Terminology". In: *lEEE Standard* (1990), pp. 610–12 (cit. on p. 8).

[89] T. Ramalingom, Krishnaiyan Thulasiraman, and Anindya Das. "Context Independent Unique State Identification Sequences for Testing Communication Protocols Modelled as Extended Finite State Machines". In: *Computer Communications* 26.14 (2003), pp. 1622–1633. DOI: 10.1016/S0140-3664(03)00116-6 (cit. on p. 38).

[90] M.S. Raunak, Christian Murphy, and Bryan O'Haver. *An Empirical Study of Off-by-One Loop Mutation*. Tech. rep. University of Pennsylvania - Department of Computer and Information Science, 2015 (cit. on p. 9).

[91] Florian Rosenberg and Schahram Dustdar. "Business Rules Integration in BPEL - A Service-Oriented Approach". In: *7th IEEE International Conference on E-Commerce Technology (CEC 2005), 19-22 July 2005, München, Germany*. IEEE Computer Society, 2005, pp. 476–479. DOI: 10.1109/ICECT.2005.25 (cit. on pp. 2, 13).

[92] Florian Rosenberg and Schahram Dustdar. "Design and Implementation of a Service-Oriented Business Rules Broker". In: *7th IEEE International Conference on E-Commerce Technology Workshops (CEC 2005 Workshops), 19 July 2005, München, Germany*. IEEE Computer Society, 2005, pp. 55–63. DOI: 10.1109/CECW.2005.10 (cit. on pp. 2, 12).

[93] Ronald G. Ross. “The Business Rules Manifesto”. In: *Business Rules Group* 2 (2003) (cit. on p. 12).

[94] Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. “Evaluating Test Suites and Adequacy Criteria Using Simulation-Based Models of Distributed Systems”. In: *IEEE Transactions on Software Engineering* 34.4 (2008), pp. 452–470. DOI: 10.1109/TSE.2008.33 (cit. on p. 14).

[95] August-Wilhelm Scheer and Markus Nüttgens. “ARIS Architecture and Reference Models for Business Process Management”. In: *Business Process Management, Models, Techniques, and Empirical Studies*. Ed. by Wil M. P. van der Aalst, Jörg Desel, and Andreas Oberweis. Vol. 1806. Lecture Notes in Computer Science. Springer, 2000, pp. 376–389. DOI: 10.1007/3-540-45594-9_24 (cit. on p. 12).

[96] Kurt Schelfthout. *FsCheck*. 2007. URL: https://github.com/fscheck/FsCheck (visited on 04/05/2016) (cit. on pp. 12, 16).

[97] Emil Sekerinski and Kaisa Sere. “A Theory of Prioritizing Composition”. In: *The Computer Journal* 39.8 (1996), pp. 701–712. DOI: 10.1093/comjnl/39.8.701 (cit. on p. 29).

[98] Hossain Shahriar and Mohammad Zulkernine. “MUTEC: Mutation-Based Testing of Cross Site Scripting”. In: *ICSE Workshop on Software Engineering for Secure Systems, SESS 2009, Vancouver, BC, Canada, May 19, 2009*. IEEE Computer Society, 2009, pp. 47–53. DOI: 10.1109/IWSESS.2009.5068458 (cit. on p. 9).

[99] Martin Tappler. “Symbolic Input Output Conformance Checking of Action System Models”. MA thesis. Graz University of Technology, 2015 (cit. on p. 29).

[100] Satish Thatte. “XLANG: Web Services for Business Process Design”. In: *Microsoft Corporation* 2001 (2001) (cit. on p. 13).

[101] The Business Rules Group. *Defining Business Rules - What Are They Really? Final Report, Revision 1.3*. 2000. URL: http://www.businessrulesgroup.org/first_paper/br01c0.htm (visited on 04/07/2016) (cit. on p. 12).

[102] Henry S. Thompson, David Beech, M. Maloney, and Noah Mendelsohn. *XML schema part 1: Structures, Second Edition*. Tech. rep. Massachusetts Institute of Technology - World Wide Web Consortium, 2004. URL: http://www.w3.org/TR/xmlschema-1/ (cit. on p. 36).

[103] Nikolai Tillmann and Jonathan de Halleux. “Pex - White Box Test Generation for .NET”. In: *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*. Ed. by Bernhard Beckert and Reiner Hähnle. Vol. 4966. Lecture Notes in Computer Science. Springer, 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10 (cit. on p. 16).

[104] Stefan Tiran. *The Argos Manual*. Tech. rep. Graz University of Technology - Institute for Software Technology, 2012 (cit. on p. 29).

[105] Jan Tretmans. “Test Generation with Inputs, Outputs and Repetitive Quiescence”. In: *Software - Concepts and Tools* 17.3 (1996), pp. 103–120 (cit. on pp. 30, 31).

[106] Jan Tretmans and Hendrik Brinksma. “TorX: Automated Model-Based Testing”. In: *First European Conference on Model-Driven Software Engineering*. Ed. by A. Hartman and K. Dussa-Ziegler. Dec. 2003, pp. 31–43 (cit. on p. 8).

[107] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123725011 (cit. on pp. 1, 7, 14).

[108] Mark Utting, Alexander Pretschner, and Bruno Legeard. “A Taxonomy of Model-Based Testing Approaches”. In: *Softw. Test., Verif. Reliab.* 22.5 (2012), pp. 297–312. DOI: 10.1002/stvr.456. URL: https://doi.org/10.1002/stvr.456 (cit. on pp. 7, 38).

[109] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer". In: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Vol. 4949. Lecture Notes in Computer Science. Springer, 2008, pp. 39–76. DOI: `10.1007/978-3-540-78917-8_2` (cit. on p. 7).

[110] Scott Wlaschin. *Choosing Properties for Property-Based Testing*. 2014. URL: `https://FsharpForFunAndProfit.com/posts/property-based-testing-2/` (visited on 04/06/2016) (cit. on pp. 10, 16).

[111] W. Eric Wong and Aditya P. Mathur. "Reducing the Cost of Mutation Testing: An Empirical Study". In: *Journal of Systems and Software* 31.3 (1995), pp. 185–196. DOI: `10.1016/0164-1212(94)00098-0` (cit. on p. 9).

[112] Qian Yang, J. Jenny Li, and David M. Weiss. "A Survey of Coverage-Based Testing Tools". In: *The Computer Journal* 52.5 (2009), pp. 589–597. DOI: `10.1093/comjnl/bxm021` (cit. on p. 13).