Ing. Dipl.-Ing. Richard Schumi, BSc

Predicting and Testing System Response-Times with Statistical Model Checking and Property-Based Testing

DOCTORAL THESIS

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr. Bernhard K. Aichernig

Institute of Software Technology (IST)
Graz University of Technology, Austria

External Reviewer and Examiner
Prof. John Hughes, Ph.D.
Chalmers University of Technology, Gothenburg, Sweden

Graz, October 2018

AFFIDAVIT

Ing. Dipl.-Ing. Richard Schumi, BSc

Vorhersage und Testen von Systemantwortzeiten mit Statistical Model Checking und Property-Based Testing

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Betreuer
Ao.Univ.-Prof. Dipl.-Ing. Dr. Bernhard K. Aichernig

Institut für Softwaretechnologie
Technische Universität Graz, Österreich

Externer Gutachter und Prüfer
Prof. John Hughes, Ph.D.
Chalmers University of Technology, Göteborg, Schweden

Graz, Oktober 2018

Abstract

In recent years, software systems have become increasingly omnipresent. Especially since the Internet of Things is spreading into the everyday life of millions of people. However, these systems often fail to satisfy the requirements of the users, both in terms of functionality and performance. Hence, in order to increase user satisfaction, it is essential to perform a rigorous quality-assurance process, most importantly in the form of software testing.

A testing technique that became popular in recent years is property-based testing. This technique relies on randomly generated test data in order to check if functions- or systems-under-test work as expected. Moreover, it can perform a test-case generation that is based on a behavioural model of a system. The definition of such a model normally requires a high manual effort. To deal with this issue, we present a testing technique that works with business-rule models that are existing system artefacts. Hence, there is no need for a manual model definition. We apply this method in order to find bugs, to increase the confidence in the functionality of the system, and also to produce log data, which we apply for performance testing.

Performance testing comprises a group of techniques that aim to evaluate performance requirements, like responsiveness or scalability of a system. It usually involves a high number of tests that are directly executed on a system, which becomes especially cumbersome, when different usage scenarios should be considered. Therefore, we propose a model-based method that works with a fast simulation to predict the expected response times for users.

First, we run property-based testing concurrently to obtain log data from simultaneous system interactions. Based on this data, we learn a stochastic model that we apply for statistical model checking in order to receive predictions with a certain confidence. Moreover, we propose an efficient evaluation technique for such predictions. By performing hypothesis testing on the system, we can check the accuracy of our model with fewer samples than needed for the model simulation.

Our method is realised in a property-based testing tool that we have enhanced with algorithms from statistical model checking. This tool allows both, simulating stochastic models and testing the simulation results directly on a system. We demonstrate the feasibility with an industrial case study of a web-service application and by performing a comparison of two protocol implementations from the Internet of Things.

Keywords: Property-Based Testing, Statistical Model Checking, Model-Based Testing, Performance, Response Time, Latency, Web-Service Application, Business-Rule Models, Internet of Things, MQTT, FsCheck.

Kurzfassung

In den letzten Jahren sind Softwaresysteme zunehmend allgegenwärtig geworden, besonders weil das Internet der Dinge in den Alltag von Millionen einfließt. Diese Systeme erfüllen jedoch oft nicht die Anforderungen der Benutzer, sowohl hinsichtlich der Funktionalität als auch der Performance. Um die Benutzerzufriedenheit zu erhöhen, ist es daher wichtig, eine strenge Qualitätssicherung durchzuführen, vor allem in Form von Softwaretests.

Eine Testtechnik, die in letzter Zeit populär wurde, ist Property-Based Testing. Diese Technik beruht auf zufällig generierten Testdaten, die verwendet werden, um zu überprüfen, ob bestimmte Funktionen oder Systeme wie erwartet funktionieren. Darüber hinaus unterstützt diese Technik eine Testfallgenerierung basierend auf einem Verhaltensmodell eines Systems. Die Definition eines solchen Modells erfordert normalerweise einen hohen manuellen Aufwand. Um diesem Problem zu begegnen, stellen wir eine Testmethode vor, die mit Business-Rule-Modellen arbeitet, welche bereits vorhandene Systemkomponenten sind. Daher ist keine manuelle Modelldefinition erforderlich. Wir setzen diese Methode ein, um Fehler zu finden, um das Vertrauen in die Funktionalität des Systems zu erhöhen, und um Log-Daten zu erzeugen, die wir für Performance-Tests verwenden.

Performance-Testen umfasst eine Gruppe von Techniken, die darauf abzielen, Leistungsanforderungen wie Systemreaktionsfähigkeit oder Skalierbarkeit zu bewerten. Dies erfordert normalerweise eine große Anzahl von Tests, die direkt auf einem System ausgeführt werden, und wird besonders aufwendig, wenn verschiedene Anwendungsszenarien betrachtet werden. Deshalb präsentieren wir eine modellbasierte Methode, die mit einer schnellen Simulation die erwarteten Antwortzeiten für Benutzer vorhersagt.

Zuerst wenden wir Property-Based Testing in mehreren nebenläufigen Prozessen an, um Log-Daten von simultanen Systeminteraktionen zu erhalten. Basierend auf diesen Daten lernen wir ein stochastisches Modell, das wir für Statistical Model Checking verwenden, um Vorhersagen mit einer gewissen Konfidenz zu erhalten. Darüber hinaus schlagen wir eine effiziente Evaluierungstechnik für solche Vorhersagen vor. Indem wir Hypothesentests am System durchführen, können wir die Genauigkeit unseres Modells mit weniger Proben, als für die Modellsimulation erforderlich sind, überprüfen.

Unsere Methode wird in einem Property-Based Testing-Tool realisiert, das wir mit Algorithmen für Statistical Model Checking erweitert haben. Dieses Tool ermöglicht sowohl das Simulieren von stochastischen Modellen als auch das direkte Testen der Simulationsergebnisse auf einem System. Wir demonstrieren die Machbarkeit mit einer industriellen Fallstudie einer Web-Service-Anwendung und mit einem Vergleich zweier Protokollimplementierungen aus dem Internet der Dinge.

Schlagworte: Property-Based Testing, Statistical Model Checking, Modellbasiertes Testen, Performance, Antwortzeit, Latenz, Web-Service-Anwendungen, Business-Rule-Modelle, Internet der Dinge, MQTT, FsCheck.

Acknowledgements

Many people helped me during the course of my Ph.D. studies and supported my research that led to this thesis. I am thankful to all of them and want to mention my biggest supporters.

Most importantly, I am grateful to my supervisor Bernhard K. Aichernig, who inspired me to start with testing software and systems. Moreover, he was a great mentor during all these years, and he taught me various skills that were necessary for writing this thesis.

I would like to thank my co-authors and project partners, who helped me to write the papers that form the basis of this thesis and who supported me in carrying out the experiments. Especially, I want to mention the following people at AIT: Priska Bauerstätter, Willibald Krenn, Cristinel Mateis, Rupert Schlick, and at AVL, Elisabeth Jöbstl, Severin Kann, Robert Korošec, Christoph Schwarz, and Manfred Uschan.

Furthermore, I am grateful to my colleagues Christian Burghard and Martin Tappler, and my former colleagues Florian Lorber and Stefan Tiran, for their valuable reviews and comments that helped to improve the quality of my work. I would like to thank my external examiner John Hughes for reviewing this thesis and for serving as an external examiner.

Last but not least, I would like to express my gratitude to my family and friends for their support during my studies, and because they have accompanied me on the path that led to this thesis.

The research leading to this thesis was funded by the Austrian Research Promotion Agency (FFG), project number 845582, "Trust via cost function driven model based test case generation for non-functional properties of systems of systems" (TRUCONF) and also by Graz University of Technology, LEAD project "Dependable Internet of Things in Adverse Environments".

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Abbreviations

CUSUM	Cumulative Sum
DB	Database
EFSM	Extended Finite State Machine
ICM	Incident Manager
IoT	Internet of Things
MBT	Model-Based Testing
MLR	Multiple Linear Regression
MQTT	Message Queuing Telemetry Transport
OLS	Ordinary Least Squares
PBT	Property-Based Testing
REM	Rule-Engine Models
SMC	Statistical Model Checking
SPRT	Sequential Probability Ratio Test
STA	Stochastic Timed Automata
SUT	System-Under-Test
TA	Timed Automata
TEM	Test Equipment Manager
TFMS	Testfactory Management Suite
TFS	Test Factory Scheduler
TOM	Test Order Manager

1 Introduction

1.1 Motivation

Software systems are becoming increasingly complex, but they still have to satisfy various user expectations. One of the major user demands is that a software system should provide the expected functionality. However, besides this functional aspect, a system should also provide acceptable performance, i.e., it should be fast enough so that the users do not have to wait too long. It has been shown that users become increasingly dissatisfied, when a system has long response times [86]. This can lead to a point where users are not willing to reuse a system. Moreover, studies suggest that response times have a strong influence on the profit of Internet companies, e.g., adding only 100 ms delay can drop sales in e-commerce by 1% [148].

In order to verify the fulfilment of such functional and non-functional requirements, rigorous quality-assurance methods are usually performed during software development. One of the most common approaches to check functional requirements, are manually written unit tests. Such unit tests usually contain specific test data in order to evaluate a certain functionality of a system. However, their definition requires a high amount of manual effort, and they only cover limited aspects of the system functionality. Testing is often the largest phase in a software-development project [134] and it can reach over 50% of the project time and also 50% of the costs [132]. Moreover, it has been shown that only about 55 to 60% of the logic paths of a software system are covered by manually written test cases, when no automated coverage analysis tools are applied during the testing phase [70], and that the automation of the test-case generation can significantly increase the coverage [65].

Property-based testing [47], which is a flexible random testing techniques, can overcome some of these issues. It can reduce the manual effort by automatically generating test data, and it can produce tests according to a model of the system behaviour, which helps to cover various system functionalities. Usually, such models have to be implemented manually, but we present a method that takes a system component as a basis for a model. This provides an even higher degree of automation.

In order to evaluate performance requirements of a system, usually techniques like performance testing, stress testing, or load testing are performed [130]. Such techniques analyse the responsiveness and scalability of a system, e.g., they check if an increasing number of users is supported. A disadvantage of these approaches is that they usually require many tests to be executed directly on the real system. This is especially cumbersome when various usage scenarios should be considered. Fortunately, there exist alternative solutions, like statistical model checking, that can accelerate such analyses by simulating a model to estimate the performance of a system.

Statistical model checking [2] is an approximate simulation-based method that can calculate probabilities or perform hypothesis tests of models or systems. We apply statistical model checking to predict the probability that a system meets certain response-time thresholds under various usage scenarios. Moreover, we illustrate how such predictions can be efficiently tested on real systems with hypothesis testing. This should help to increase the confidence in the expected performance of a system.

1.2 Property-Based Testing

Property-based testing (PBT) [47] is a random testing technique that tries to falsify a given property. A property is defined for a function- or system-under-test (SUT) and it describes its expected behaviour. In order to test such a property, a PBT tool generates random inputs for the function or SUT and checks if the expected behaviour is observed. PBT can be performed with different types of properties. Simple algebraic properties are predicates that can, e.g., check the output of a function-under-test. For more complex evaluations, PBT also supports model-based testing (MBT) [179], where the expected behaviour of a function or SUT is described with a behavioural model. Such models are usually in the form of a state machine. In order to perform PBT with such models, it is necessary to implement a state-machine specification that defines the connection between the model and the SUT. This specification comprises functions to initialise the model and the SUT, commands that define the possible actions, and a generator that produces the next command for the current state of the model. A command has a precondition that specifies when it is enabled, a postcondition that describes the expected behaviour, and functions to execute the model and the SUT. In order to evaluate such a specification, a PBT tool produces random command sequences, executes them on the SUT and checks the postconditions. A command sequence can also incorporate generated test data (e.g., form data). An advantage of PBT is that it facilitates the generation of complex test data. It provides default generators (for standard data types) that can be nested, extended, combined, etc. to form custom generators.

We can characterise MBT with PBT based on a taxonomy of Utting et al. [180] as illustrated in Figure 1.1, where the applicable categories are coloured in blue. PBT supports a variety of input-output models that may have non-deterministic characteristics and may include timing behaviour. It has a pre-post modelling paradigm that is transition-based and can include stochastic choices. The test-case generation is usually random (with a uniform distribution), but other stochastic distributions can also be applied for the test selection. The test execution is usually offline, i.e., a test case is only executed on a system, after it was generated [5].

PBT is especially convenient for applications, where complex test data is needed. Consequently, it has been applied for web-service and protocol testing [22, 67, 118, 144]. Moreover, PBT is helpful for performance testing as its random input generation enables an analysis with a variety of different inputs. For example, we apply PBT for load testing [59, 127], where we run several test-execution processes concurrently in order to simulate the behaviour of various user populations and to find out the limits of a given SUT. More details about PBT are given in Section 2.1.

1.3 Statistical Model Checking

Statistical model checking (SMC) [2] is a simulation-based method that can answer both quantitative and qualitative questions. For example, questions like "What is the probability that a model satisfies a property?" or "Is this probability greater, or below a certain threshold?". More concretely, a question might be "How likely will a system respond within 100 ms given a specific usage scenario?" [6].

In order to answer such questions, a statistical model checker will simulate the model (or system) and check if the property is satisfied for this simulation. A simulation of the model represents a sample. SMC algorithms either calculate the number of needed samples or a stopping criterion, which describes when they can finish with a required confidence [6].

The simplest SMC algorithm is a Monte Carlo simulation. For this algorithm, the model is executed with a fixed number of samples and the portion of the samples that satisfy a given property gives us an estimate for the probability that the property holds.

Another common SMC algorithm is the sequential probability ratio test (SPRT) [187]. This sequential algorithm is a form of hypothesis testing, where either a null or an alternative hypothesis is accepted. We have a stopping criterion based on given error parameters. As long as the stopping criterion is not true, we are in an indifference region and have to continue sampling. We can stop, when we are outside this region, i.e., when the stopping criterion

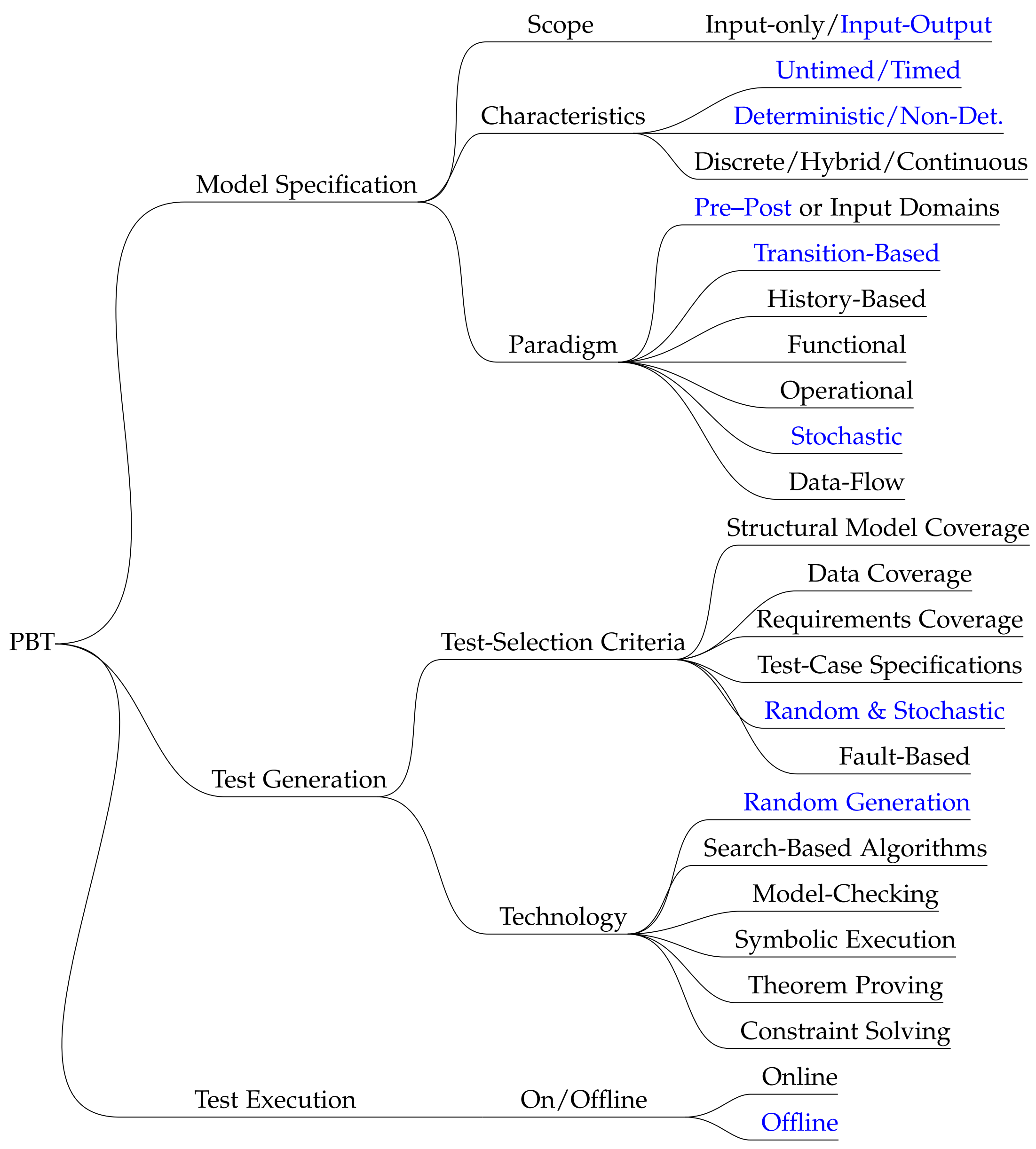


Figure 1.1: Taxonomy of MBT in relation to PBT based on Utting et al. [180].

holds. When the upper or lower bound of the region was reached, then we accept either the null or alternative hypothesis. More details about SMC algorithms are given in Section 2.2.

1.4 Research Context

1.4.1 Research Projects

The work of this thesis has been conducted within two research projects. The first project and major project of this thesis was TRUCONF (Trust via cost function driven model based test case generation for non-functional properties of systems of systems).[1] This research project was joint work between the Austrian Institute of Technology (AIT)[2], AVL List GmbH[3] and

[1]`http://truconf.ist.tugraz.at` (visited on 2018-09-19)
[2]`https://www.ait.ac.at` (visited on 2018-09-19)
[3]`https://www.avl.com` (visited on 2018-09-19)

the Graz University of Technology (TU Graz) and it was funded by the Austrian Research Promotion Agency (FFG). The aim of the project was to verify the reliability of systems-of-systems, not only in terms of the functionality of the systems, but also the non-functional reliability, like the performance of the systems. TRUCONF intends to combine cost-function learning and model-based testing in order to support this kind of verification. Moreover, it was planned to introduce new modelling notations/languages in order to increase the accessibility for developers from industry.

The second project of this thesis is called Dependable Things (Dependable Internet of Things in Adverse Environments)[4] and it is funded by TU Graz. It is an ongoing flagship project that encourages the cooperation among research groups within TU Graz. Therefore, the project is joint work of ten researchers from different institutes and research fields of TU Graz, like electrical and information engineering, and computer science. The aim of this project is to increase the dependability of "Smart Things", which form the basis for the Internet of Things (IoT). The IoT is becoming increasingly popular, as technologies, like smart homes, connected cars and so on, are reaching the end user market. However, dependability aspects that comprise reliability, safety, and security are still not verified thoroughly enough. Hence, the Dependable Things project intends to eliminate this problem in order to increase the trust in the IoT.

Both these projects contributed a case study that was dealt with during this thesis and is described below.

1.4.2 Case Studies

TFMS. *This description is taken from our previous work [5, 9].*

Our industrial partner from the automotive industry (AVL) provided us with a web-service application called testfactory management suite (TFMS). This industrial application was used as the main case study of the TRUCONF project. The application originates from the automotive domain and is a workflow tool that supports the process of instrumenting and testing automotive power trains – a core business of AVL. TFMS captures test-bed data, activities, resources, and workflows. A variety of activities can be realised with the system, like test definition, planning, preparation, execution, data management, and analysis.

The application is intended for various kinds of automotive test beds for car components, like engines, gears, power trains, batteries for electric cars or entire cars. For instance, for testing an engine it is mounted to a pallet and also different test equipment is attached. The selection of the test equipment depends on the specific use case. Typical test equipment for an engine might be a measurement device for the power output or the fuel consumption. After a pallet is configured, it is moved to the test bed, where all devices are connected and a test is performed. TFMS manages all steps and devices required by such a workflow, which is also called a test order. It allows the scheduling of car components that need to be tested, the selection of required test equipment, the definition of the required wiring for the equipment, and the planning of the sequence of all tasks at a test bed. Moreover, customer specific requirements, like additional management steps or custom restrictions, can be freely configured via business rules.

The system has a client-server architecture which is illustrated in Figure 1.2. The "TFMS Server" is the central component of the system. This server is hosted in Microsoft's IIS (Internet Information Services) and provides several simple object access protocol (SOAP) web services, which are described via the web services description language (WSDL). For data storage, MongoDB[5] is used. TFMS offers different types of client applications: one to collect

[4] `https://www.tugraz.at/projekte/dependablethings` (visited on 2018-09-19)
[5] `https://www.mongodb.com` (visited on 2018-09-19)

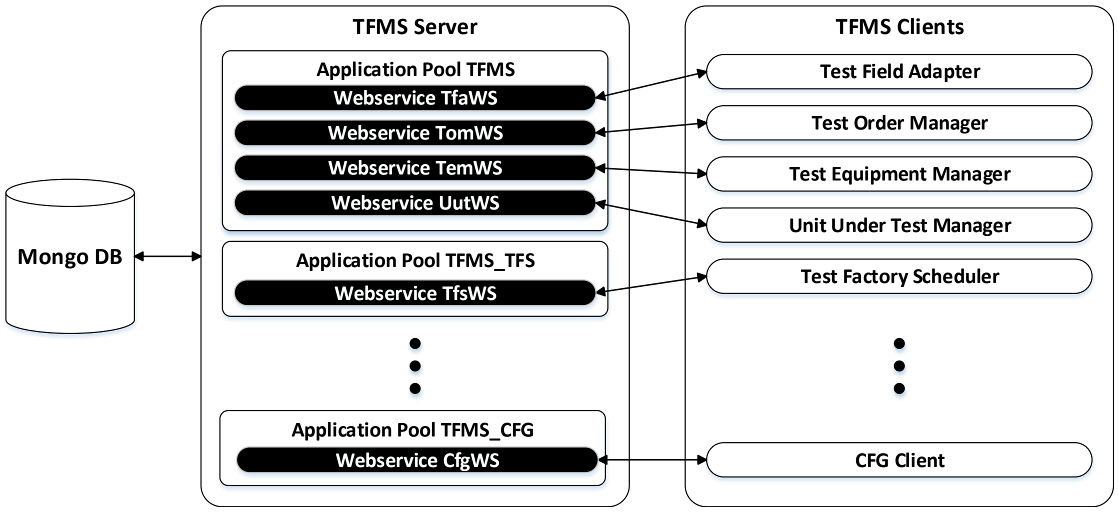


Figure 1.2: Client-server architecture of the TFMS.

data from the test beds, several office clients for different management activities (e.g., test order management) and a scheduler to plan the execution of test activities on the test beds.

TFMS is highly configurable and offers an own client application for server configurations (CFG Client). The web services are driven by a custom implementation of business rules. A rule engine takes this business logic in the form of business-rule models and interprets them, which defines the control-flow of the application. The system consists of multiple modules corresponding to the mentioned clients. Modules can be seen as groups of functionality, and they consist of multiple business-rule models which describe what tasks can be performed by a user and how they look like, e.g., what data can be modified. Only one business-rule model can be active within one client application and it determines, which forms can be opened in the current state of the system.

A business-rule model is a state machine defining the behaviour of the business objects, so called TFMS Objects. A TFMS Object class describes objects of our application domain, like test equipment or test orders. Each object has a state, an identifier, attribute values/data and is stored in the database of our SUT. TFMS works task-based. Tasks represent the behaviour, i.e., the actions or events a user may trigger, e.g., creating or editing TFMS objects. Example business-rule models and a description of tasks and subtasks are presented in Chapter 3.

TFMS is a critical software, because it is essential to efficiently operate test beds. It is deployed at various customers where it is running under different hardware and network settings. Moreover, it is applied for several application fields and under varying usage conditions, i.e., with several users and different user types. It is important for AVL that the system is fast enough to satisfy even high numbers of (concurrent) users. Hence, in this work we are investigating the performance of TFMS for various usage scenarios and also for different deployments.

MQTT. A case study that we considered within the Dependable Things project was the Message Queuing Telemetry Transport (MQTT) protocol [28], which is an important communication standard within the IoT. This protocol follows a publish-subscribe pattern and allows clients to subscribe to topics that are maintained on a central broker. A subscriber can, e.g., be a personal computer, a smart phone or any device that should react to or monitor published messages. Clients that publish messages are often sensors, e.g., a temperature sensor in a smart home, but also various other devices may publish messages.

A client can publish a message to a topic by sending the message to the broker. The broker will then distribute this message to the subscribed clients. Figure 1.3 illustrates the

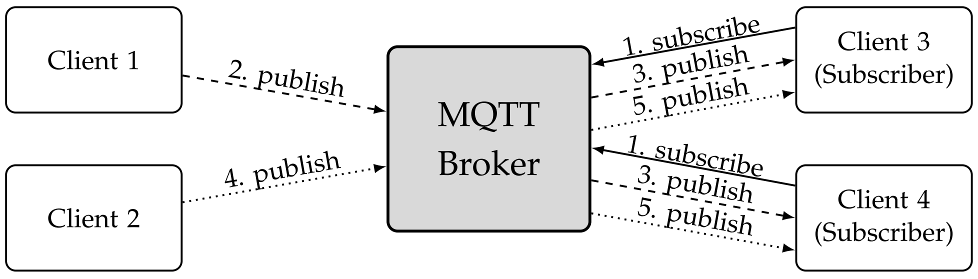


Figure 1.3: Overview of the interactions of an MQTT setup (based on Fujita et al. [68]).

interactions within an MQTT network under the assumption that we only have one topic. (This figure is based on a concept diagram from Fujita et al. [68].) It can be seen that clients first subscribe to the given topic by sending a subscribe message to the broker (1). Then, when a client publishes a message to this topic by sending the message to the broker (2), it is forwarded to the subscribed clients (3). A further publish message (4) of another client is also sent to the same subscribed clients (5) by the broker.

There exist various client and broker implementations that follow the MQTT standard. However, it is not clear if all of them function according to the standard, and which implementation is the fastest under a specific usage condition. In this work, we are interested in the performance of specific MQTT broker implementations, i.e., we test the latencies from the perspective of a client. Additionally, we can also test the functionality and robustness of these implementations.

1.5 Problem Statement and Research Questions

PBT is a flexible testing technique, especially its model-based testing feature is useful for many applications. However, like most model-based testing approaches, it still requires manual effort for the definition of a model. Various related approaches showed how some web-service descriptions can facilitate the model definition [62, 64, 109, 116], but they do not construct a complete model with these descriptions.

The challenge of the high modelling effort and the fact that our SUT was driven by a business-rule engine led to the following research question:

RQ1: Can business-rule models be applied as test models for property-based testing in order to perform load testing and also to find bugs?

- RQ1.1: What kind of bugs and issues can be found?
- RQ1.2: What are the benefits and drawbacks compared to conventional model-based testing?
- RQ1.3: What are adequate test-case generation strategies for such models?

Another important challenge of this thesis was performance testing. Performance testing is a difficult task, but it is important to ensure that users are satisfied and do not have to wait too long for a system response. It becomes especially challenging when various usage scenarios should be considered. There are existing methods that directly test a system, like load testing, and estimation techniques that simulate the performance with a model. However, they both suffer from certain disadvantages. On the one hand, a high number of tests that are directly executed on the system is needed, and on the other hand, the accuracy of the models is questionable. In order to cope with these problems, we propose a combination of these techniques. As explained earlier, we apply PBT for load testing, because it facilitates the generation of test data, and we apply SMC for the model simulation. However, it was not clear if these methods can work together, which led to the following research question:

RQ2: Is it possible to perform statistical model checking within a property-based testing tool?

- RQ2.1: What are the differences to conventional statistical model checking?
- RQ2.2: What kind of questions can be answered?

Given a PBT tool with integrated statistical model checking, we wanted to assess the performance of the aforementioned systems (Section 1.4.2), which resulted in the next research question:

RQ3: Can a property-based testing tool be applied to predict the probability that a system satisfies certain response-time thresholds for specific user populations?

- RQ3.1: What kind of user populations can be simulated?
- RQ3.1: How fast is the prediction?

The fact that predictions are often inaccurate led to the final research question:

RQ4: Is it possible to verify these predictions about the expected response time by directly testing a system-under-test?

- RQ4.1: What is an efficient way to test the predictions?
- RQ4.2: How accurate are the predictions?

1.6 Research Methodology

The research of this thesis consists of exploratory, constructive, and empirical steps. First, we preformed a literature survey of the state-of-the-art in order to find existing approaches that are related to our research problem. This exploratory research is shown in Chapter 8. During this stage, we noticed that the exiting approaches have some problems as explained in Section 1.5.

In order cope with these problems, we applied constructive research. For this, we developed a tool for business-rule testing (Chapter 3), extended this tool with SMC algorithms (Chapter 4) and evaluated our implementation by repeating case studies from the literature and by a performance comparison with existing tools. Furthermore, we developed a simulation and testing method that applies our tool and performs a combination of load testing and a model-based performance prediction. This method supports fast statistical simulations of a model and also statistical testing of a real system, both within one tool.

We applied empirical research for the evaluation of this method. It was applied to our two case studies (Chapter 7) and we tried to falsify our hypotheses that the predicted performance of the SUT is at least as good as the model. Moreover, we evaluated the run times of the model simulation and of the hypothesis tests of the SUT in order to highlight the efficiency of our proposed method.

The phases and the techniques that were applied for our method are illustrated in Figure 1.4, which outlines the overall process presented in this thesis.

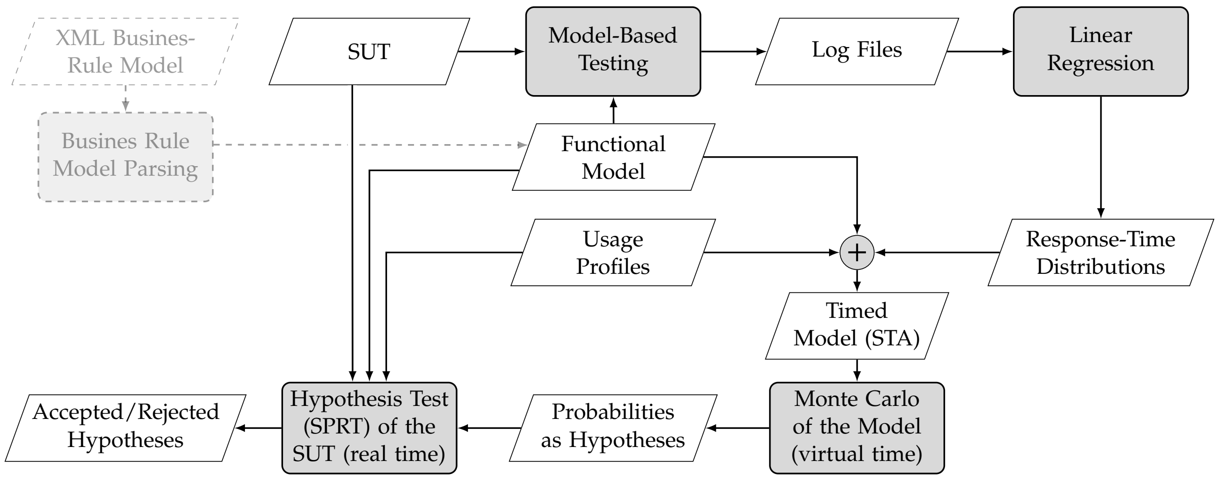


Figure 1.4: Overview of the data flow of our method.

1. First, we take business-rule models (XML files) from our SUT as input and parse them in order to obtain a functional model. Alternatively, the functional model can be defined manually, which was, e.g., the case for the MQTT case study.
2. With this functional model, we perform model-based testing within a PBT tool to produce log data. We run several testing processes concurrently and capture log files that include response times (or latencies) of simultaneous requests (or messages). Note that with this concurrent test execution, we want to measure the response time for different system loads. This means that this testing phase is a form of load testing.
3. Next, we perform a linear regression to learn response-time distributions from the log data.
4. Then, we integrate these distributions and stochastic usage profiles into the functional model. This gives us a combined model with the semantics of stochastic timed automata (STA) [25].
5. This combined model is then executed with a Monte Carlo simulation in order to evaluate the probability that each user within a given population receives responses within a certain time threshold.
6. Finally, we evaluate the predicted probabilities with hypothesis testing, namely with the sequential probability ratio test (SPRT) [187], because this method usually requires fewer samples than a Monte Carlo simulation.

1.7 Thesis Statements

In the following, we introduce the main statements that summarise the developed techniques of this thesis and that should also serve as take-home messages.

The application of business-rule models for model-based testing makes sense for finding bugs and also for load testing. It also supports a higher degree of automation since no manual model definition is needed, like it is usually the case for model-based testing.

The application of a functional model for model-based testing enables the extension of the functional model to a model with non-functional behaviour. This can, e.g., be done by learning non-functional aspects, like the response time, from log data collected during the execution of model-based testing.

Such an extended model enables a prediction of non-functional properties with a Monte Carlo simulation and these predictions can be efficiently verified with hypothesis testing, since this usually can be done with fewer samples.

1.8 Contributions

This thesis presents the following major contributions:

- We introduced a model-based testing approach that works with business-rule models and can perform load testing and is also able to find bugs.
- Another contribution is the extension of a PBT tool with SMC algorithms. This extension supports the statistical analysis of models and systems. Moreover, it supports conformance testing of a stochastic faulty system by comparing it to an ideal model. The extension was made for the PBT tool FsCheck, and we made it open-source in order to contribute to the community.
- We illustrated how a functional model can be applied for model-based testing to produce log data for learning non-functional aspects. Additionally, we used these non-functional aspects in order to extend the initial functional model.
- We demonstrated how to apply a learned timed model to simulate the expected response times of certain usage scenarios.
- Furthermore, we introduced an efficient evaluation method that can test the accuracy of our learned model by checking the model prediction with hypothesis testing. This allows us to identify usage scenarios that show a poor performance, which is necessary, because we want to maximise the user satisfaction.
- Another contribution is the evaluation of our simulation and verification method by applying it to an industrial web-service application and to open source protocol implementations.
- We introduced a further application possibility of this method, i.e., for deployment testing, where we derive hypotheses about the expected response times from a reference system to evaluate the performance of different deployments of this system with various hardware and network settings.

1.9 Publications

Most of the work of this thesis has already been published in international workshop proceedings, conference proceedings, and journals. The published papers have gone through a rigorous peer review process. One paper [9] has not yet been published, but it has been accepted with minor revisions. The following publications present the main contributions and form the basis of this thesis:

A-MOST 2016 [4]. This paper introduced our testing approach that works with business-rule models. The paper was mostly written by me, under the supervision of Bernhard K. Aichernig, who wrote some small passages and polished the text. I performed all experiments. The paper was published in the proceedings of the 12th Workshop on Advances in Model Based Testing (A-MOST) and I presented it at this workshop in Chicago, USA in 2016.

MEMOCODE 2016 [7]. In this paper, we introduced SMC properties that facilitate executing SMC algorithms within a PBT tool. Moreover, we applied this method for a stochastic conformance analysis of an example implementation with stochastic faults. Most of the work was done by me, but Bernhard K. Aichernig helped with some polishing and contributed some small parts of the text to the paper. The work was published in the proceedings of the 14th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE) and I presented it in Kanpur, India in 2016.

ICST 2017 [6]. This work extended the previous paper, where we introduced SMC properties. In this paper, we presented some optimisations for the evaluation of stochastic models and we showed an extensive evaluation of this approach. This evaluation was done by repeating several case studies from the SMC literature. I did most of the work and Bernhard

K. Aichernig contributed small parts of the text, proofread and revised the paper. I presented the paper at the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST) in Tokyo, Japan, in 2017.

ICTSS 2017 [162]. We introduced our model-based simulation and verification method in this paper, and we also presented an evaluation of this method with our TFMS case study. First, we performed model-based testing in order to obtain log data, which was then applied to learn response-time distributions with a linear regression. These distributions allowed us to simulate the expected response times of stochastic usage profiles, and we verified the simulation results with hypothesis testing on the real system. The paper was joint work with Priska Bauerstätter (maiden name Priska Lang), Bernhard K. Aichernig, Willibald Krenn, and Rupert Schlick. I wrote most of the content of the paper, developed the theory, and performed the experiments. Priska Bauerstätter did the linear regression and wrote the corresponding sections. The other authors proofread and polished the content. The paper was published in the proceedings of the 29th IFIP International Conference on Testing Software and Systems (ICTSS) and I presented it in St-Petersburg, Russia in 2017.

SoSyM 2017 [5]. In this work, we extended our previous paper [4], where we introduced our MBT approach for business-rule models. We added additional formalisations and algorithms for PBT and for the translation of the TFMS business-rule models to models for PBT. Moreover, we presented an additional case study. I produced most of the content and did the experiments. Bernhard K. Aichernig helped with the formalisations, contributed small texts, and made some textual improvements. The paper was published with open access in the International Journal on Software and Systems Modeling (SoSyM).

SQJO [9]. This journal paper is an extension of our ICTSS paper [162]. We added a second case study, and we also included an extensive description of the response-time learning phase. The paper was joined work with Bernhard K. Aichernig, Priska Bauerstätter, Elisabeth Jöbstl, Severin Kann, Robert Korošec, Willibald Krenn, Cristinel Mateis, and Rupert Schlick. I did most of the paper writing. The authors from AVL contributed a description of the TFMS, helped with some revising, and Severin Kann executed the experiments in a distributed environment at AVL. Cristinel Mateis from AIT contributed the response-time learning technique and wrote the corresponding sections. The other authors proofread the paper and made some textual improvements.

SETTA 2018 [12]. This work presents an application of our model-based simulation and verification method for deployment testing. We learned a model about the expected response times of a reference system in order to derive hypothesis that we applied to analyse the performance of deployments of this system that have a different hardware or network setting. The paper was joint work with Bernhard K. Aichernig and Severin Kann. I wrote most of the content, designed the experiments and did the necessary implementations. Bernhard K. Aichernig made a few corrections. Severin Kann helped to run the experiments in an AVL environment and contributed some small descriptions of our SUT. The paper was published in the proceedings of the 4th Symposium on Dependable Software Engineering Theories, Tools and Applications (SETTA) and I presented it in Beijing, China in 2018.

QEST 2018 [3]. In this paper, we applied our model-based simulation and verification method, which we presented in the ICTSS paper [162], to MQTT implementations in order to evaluate their performance. Most of the work was done by me under the supervision of Bernhard K. Aichernig, who revised the paper. I presented this paper at the 15th International Conference on Quantitative Evaluation of SysTems (QEST) in Beijing, China in 2018.

The following publication is also related to this thesis, but is not part of the main contributions:

A-MOST 2017 [10]. We presented a PBT approach with an external test-case generator in this paper. We compared the random generation technique of classical PBT with a more targeted test-case generation that is based on model-based mutation testing. The paper was joint work with Bernhard K. Aichernig and Silvio Marcovic. I wrote some contents of this paper, Silvio Marcovic supplied most of the content and performed the experiments and Bernhard K. Aichernig proofread and polished the paper. The paper was published in the proceedings of the 13th Workshop on Advances in Model Based Testing (A-MOST) and I presented it at this workshop in Tokyo, Japan in 2017.

1.10 Structure

The rest of this thesis is structured as follows: Chapter 2 explains the necessary background information for this thesis, like PBT, SMC, linear regression and stochastic timed automata. In Chapter 3, we present our model-based testing approach with business-rule models. Next, Chapter 4 illustrates how we integrate SMC algorithms into a PBT tool, and we highlight the applicability of this approach with examples from the SMC literature. Then, in Chapter 5, we show how we generate log data that contains response times of simultaneous requests with MBT. Moreover, we demonstrate how we can learn response-time distribution via linear regression. Chapter 6 presents our performance prediction and testing method. We show the construction of our combined model by integrating stochastic usage profiles and the learned response-time distributions into our existing functional models. We demonstrate how these models can be executed with a Monte Carlo simulation in order to predict probabilities about the expected response times. Moreover, we illustrate an analysis of the accuracy of our models with hypothesis testing, since this can efficiently be done on a real system. Next, in Chapter 7, we evaluate our model-based simulation and verification method by performing several case studies with the aforementioned TFMS and MQTT. Chapter 8 presents work related to the main methods of this thesis. Finally, Chapter 9 gives a summary of the thesis, discusses the research questions and contributions, explains potential future work, and concludes the work.

2 Background

This chapter contains parts from our publications in the journal SoSyM 2017 [5], at ICTSS 2017 [6], at QEST 2018 and in the journal SQJO 2017 [9]. Especially, the linear regression in Section 2.3, was joint work with Cristinel Mateis from AIT.

2.1 Property-Based Testing

2.1.1 Overview

Property-based testing (PBT) [47] is a random testing technique that evaluates a system by verifying a given property. A property is a high-level specification of behaviour that should hold for a range of data points. For example, a property might state that a function should have a certain output. A test of this property is successful, when the function runs through as expected, otherwise a counterexample is returned. Simple properties can be expressed as functions with Boolean return values that are true when the property is fulfilled. These properties should hold for any input value, hence a high number of random inputs are generated for the parameters. A simple example of an algebraic property is that the reverse of the reverse of a list must be equal to the original list:

$$\forall xs \in List[T] : reverse(reverse(xs)) = xs$$

To evaluate this property, a PBT tool will invoke its built-in generator for *List*s and generate a series of random lists *xs*, execute the reverse function and evaluate the property. A tester may extend or replace the basic generators with special-purpose generators, e.g., for generating extremely long lists. Generators are type-based and provide a *sample* function for the given type. In the simplest case, a generator $Gen[A]$ for a type A provides a function $sample : () \to A$ that returns an instance of this type. For nested data types, generators take other generators as arguments. For example, a generator for $List[T]$ needs a generator for element values of type T [89, 143, 158].

Another important aspect of PBT is shrinking, which is used to find a similar simpler counterexample, when a property fails. In order to shrink a counterexample, a PBT tool searches for smaller failing counterexamples. The search method can be specified individually for different data types [89, 143, 158].

When we consider the property from above and assume a faulty reverse function, then shrinking might help us to reduce a long list with many elements to a list with just two elements that can still reproduce the same fault.

PBT constitutes a flexible and scalable testing technique, because it is random testing and it has been shown that it generates a large number of tests in reasonable time [185]. The first PBT tool was QuickCheck [47] for Haskell. There are many other tools that are based on the concepts of QuickCheck, e.g., ScalaCheck [136] or Hypothesis[6] for Python. For our approach, we work with FsCheck[7].

2.1.2 FsCheck

FsCheck is a PBT tool for .NET based on QuickCheck and influenced by ScalaCheck. Like ScalaCheck, it extends the basic QuickCheck functionality with support for state-based models. With FsCheck, properties can be defined both in a functional programming style with F#

[6] `https://pypi.python.org/pypi/hypothesis` (visited on 2018-09-19)
[7] `https://github.com/fscheck/FsCheck` (visited on 2018-09-19)

and in an object-oriented style with C#. Similar to QuickCheck, it has default generators for basic data types and more complex ones can be defined via composition. It enables a simple introduction of new data types via *Arbitrary* instances that group together a shrinker and a generator for a custom data type. This makes it possible to use variables of this data type as input for properties. New *Arbitrary* instances can be dynamically registered at run time and then the new data type can be directly used for the input-data generation. Furthermore, FsCheck has extensions for unit testing, which support a convenient definition and execution of properties like normal unit tests.

2.1.3 Model-Based Testing

As already explained in Section 1.2, PBT can be applied for model-based testing (MBT) and it supports various types of state-based models that may have non-deterministic characteristics and may include timing behaviour.

In order to perform MBT with PBT, we need a model of the SUT, an interface to the SUT, and a state-machine specification that defines the connection between the model and the SUT. The model is usually in the form of a state machine and it represents the behaviour of the system in an abstract form. A state-machine specification works with this model (and the SUT) and serves as a basis for state-machine properties that are applied to generate and execute test cases. Such a specification has to contain (1) functions to initialise the model and the SUT, (2) commands that define the possible actions and their execution on the model and the SUT, and (3) a generator that produces the next command for the current state of the model.

A command usually represents an input or action of the SUT and it describes how the model state evolves and which interactions are performed on the SUT. It comprises (1) an optional precondition that specifies when it is enabled, (2) a postcondition that performs the verification of the expected behaviour and (3) functions to execute the model and interact with the SUT. In order to evaluate such a specification, a PBT tool produces random command sequences (i.e., test cases), executes them on the SUT and checks the postconditions. A command sequence can also incorporate generated test data (e.g., form data). The data generation represents a big advantage of PBT, since it facilitates the generation of complex test data.

Moreover, the generated test cases can be minimised with the shrinking feature of PBT. For this purpose, shorter command sequences are produced. Additionally, the size of the test data that can be contained within command instances may be reduced in order to find a simpler test case that can reproduce a fault.

A more detailed formal definition of the test-case generation and execution will follow in Chapter 3.

Example. In order to demonstrate state-machine specifications, we present an example for testing a simple counter that is commonly used in the PBT community [52]. Listing 2.1 illustrates the example specification for FsCheck. The counter has three functions: (1) increment by one, (2) decrement, and (3) retrieving the current value of the counter. The model of this counter is just a normal integer representing the state of the counter. Line 1 of Listing 2.1 shows that we need to implement an *ICommandGenerator* interface of FsCheck that takes the SUT and the model as type parameters. This interface requires functions to initialise the model (Line 4) and the SUT (Line 5), and a function called *Next* (Line 8–10) that takes the current value of the model as input and returns a new generator. In this case, it is an *Elements* generator that selects one element of an array of our commands (Line 9). This array contains two command objects one for the increment *IncCmd* and one for the decrement *DecCmd*.

```
 1 public class CounterSpec : ICommandGenerator<Counter, int> {
 2
 3   //Initialisation of the model and SUT
 4   public int InitialModel{ get { return 0; }} // return zero to initilise the integer
         model
 5   public Counter InitialActual{ get { return new Counter(); }}
 6
 7   //Generator for the next Command given the state of the model
 8   public Gen<Command<Counter, int>> Next(int m) {
 9     return Gen.Elements(new Command<Counter, int>[] {new IncCmd(), new DecCmd()});
10   }
11
12   //Increment command
13   private class IncCmd : Command<Counter,int> {
14     public override int RunModel(int m) { //Executes the model that is given as input
15       return m + 1;
16     }
17     public override Counter RunActual(Counter c) { //Executes the SUT
18       c.Inc();
19       return c;
20     }
21     public override Property Post(Counter c, int m) { //Postcondition
22       return (m == c.GetValue()).ToProperty();
23     }
24     public override string ToString() {
25       return "Inc";
26     }
27   }
28   //Decrement command (DecCmd)...
29 }
```

Listing 2.1: FsCheck specification of a counter from the FsCheck website [52].

An example of a command class for an increment is shown in Line 13. As explained earlier, a command contains functions for the execution of the model (Line 14–16) and the SUT (Line 17–20). Both these functions take the current model or SUT as input and return a modified version that, e.g., has an updated internal state. In this example, we just increase the integer for the model execution and call the *Inc()* function of the counter for the execution of the SUT. The postcondition of the command is presented in Line 21–23 and it compares the state of the model with the state of the SUT. The Boolean result of this comparison is converted with the *ToProperty()* function in order to make it applicable for FsCheck. Finally, the command also contains a *ToString()* function for its output representation. Note that the similar *DecCmd* class was omitted for brevity.

FsCheck can apply such a state-machine specification as a basis for the test-case generation and also for the test execution on the SUT. An example test case for the *Counter* specification is the following:

```
Inc, Inc, Dec, Inc, Dec, Inc, Inc, Inc, Dec, Inc, Inc Inc, Dec, Dec, Inc
```

Per default, FsCheck generates 100 such test cases with increasing length. If the test cases are executed successfully, then we can be confident that our tested implementation might work as expected, but there is no guarantee. However, in case of a failure during the execution FsCheck produces a minimal counterexample. For example, if we have a faulty *Counter* implementation that fails after two increments, then FsCheck might give us a simple test case with three increments. Such a minimal counterexample is produced with shrinking, which is especially helpful when the generated test cases become long and are difficult to analyse manually.

2.2 Statistical Model Checking

Statistical model checking (SMC) [2, 198] is a simulation-based method for checking qualitative and quantitative properties of stochastic models or systems. These properties are usually defined with (temporal) logics, like with the Bounded Linear Temporal Logic (BLTL), which enables the expression of logical formulas with time bounded operators [95, 141, 146]. For example, a property might state "An SUT will eventually be in an error state, when we consider an execution time bound of 10 seconds." With SMC, we can answer questions about such properties, like "What is the probability that the model satisfies a property?" or "Is the probability that the model satisfies a property above or below a certain threshold?". In order to answer such questions, a statistical model checker produces samples, i.e., random walks on the model or system, and checks whether the property holds for these samples. Various SMC algorithms are applied in order to compute the total number of samples needed to find an answer for a specific question, or to compute a certain stopping criterion. A stopping criterion determines when we can stop sampling, because we have found an answer with a required certainty [2, 113, 115].

A big advantage of SMC is that it enables fast simulations that can easily be parallelised by generating samples on multiple machines. Moreover, it does not suffer from the state-space explosion problem like classical model-checking approaches, since the construction of the state space is not needed for simulation. Hence, it is the only option for many realistic systems [38]. For example, it has already been applied for the evaluation of protocols [41, 83] or for biological systems [49, 55].

There exist numerous tools for SMC, like UPPAAL-SMC [42] or PLASMA-lab [38, 94]. Later in Chapter 8, we will discuss further tools and their supported algorithms and also other application areas of SMC. In this section, we focus on the following algorithms common in the SMC literature [2, 113, 115].

2.2.1 Standard Monte Carlo Simulation

This is the simplest SMC algorithm. It answers quantitative questions and works as follows. First, a fixed number of samples and a property are specified by the user. Then, the statistical model checker simply generates the specified number of samples and counts for how many of them the property holds. Hence, a sample represents a Bernoulli experiment that has two possible outcomes: true if the property holds and false otherwise. Finally, the number of samples that fulfil the property divided by the total number of samples is used to estimate the probability that the model satisfies the property [38].

Next, we introduce a more sophisticated version of a Monte Carlo simulation that enables an estimation with a desired confidence and with a specific error bound.

2.2.2 Monte Carlo Simulation with Chernoff-Hoeffding Bound

The algorithm computes the required number of simulations n in order to estimate the probability γ that a stochastic model satisfies a Boolean property. The procedure is based on the Chernoff-Hoeffding bound [82] that provides a lower limit for the probability that the estimation error is below a value ϵ. Assuming a confidence $1 - \delta$ the required number of simulations can be calculated as follows:

$$n \geq \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right) \tag{2.1}$$

The n simulations represent independent and identically distributed Bernoulli random variables $X_1, \ldots, X_n$ with outcome $x_i = 1$ if the property holds for the i-th simulation run and

$x_i = 0$ otherwise. Let the estimated probability be $\bar{\gamma}_n = (\sum_{i=1}^{n} x_i)/n$, then the probability that the estimation error is below ϵ is greater than our required confidence. Formally we have:

$$Pr(|\bar{\gamma}_n - \gamma| \leq \epsilon) \geq 1 - \delta. \tag{2.2}$$

After the calculation of the number of required samples n, a standard Monte Carlo simulation is performed [113].

Note that this algorithm works, because the Chernoff bound (2.2) gives us the minimum probability that n independent and identically distributed Bernoulli random variables will only deviate from the expected value γ by a small error ϵ. Moreover, it states that more samples will increase this minimum probability (confidence) or allow us to decrease the error bound ϵ. By rearranging the Chernoff bound, we obtain the minimum number of samples (2.1) for a specific error ϵ and a confidence $1 - \delta$.

Example. This algorithm can, e.g., be applied to calculate the required number of samples to make a forecast with a poll for the approval of some political agenda [174]. In particular, we can calculate how many people have to be asked in order to obtain an estimation with a desired confidence $1 - \delta$ and an error ϵ. For example, for $\delta = 0.03$ and $\epsilon = 0.02$, we need to ask

$$n \geq \frac{1}{2 \times 0.02^2} \ln\left(\frac{2}{0.03}\right) = 5249.63 \approx 5250$$

people in order to obtain an accuracy of $\pm 2\%$. This means that our estimation will be correct in 97 times out of 100 cases. Note that other stricter versions of the Chernoff-Hoeffding bound exist [174], but we applied this version, because it was used by other SMC tools, like PLASMA-lab [113].

2.2.3 Sequential Probability Ratio Test (SPRT)

This sequential method [187] is a form of hypothesis testing that can answer qualitative questions. Given a sequence of independent and identically distributed random variables $X_1, X_2, \ldots$ with a probability density function $f(x_i, \theta)$, we want to decide, whether a null hypothesis $H_0 : \theta = \theta_0$ or an alternative hypothesis $H_1 : \theta = \theta_1$ is true for desired type I and II errors (α, β). In order to make the decision, we start sampling and calculate the log-likelihood ratio after each observation of x_i:

$$\log \Lambda_m = \log \frac{p_1^m}{p_0^m} = \log \frac{\prod_{i=1}^{m} f(x_i, \theta_1)}{\prod_{i=1}^{m} f(x_i, \theta_0)} = \sum_{i=1}^{m} \log \frac{f(x_i, \theta_1)}{f(x_i, \theta_0)} \tag{2.3}$$

We continue sampling as long as the ratio (2.3) is inside the indifference region $\log \frac{\beta}{1-\alpha} < \log \Lambda_m < \log \frac{1-\beta}{\alpha}$. H_1 is accepted when $\log \Lambda_m \geq \log \frac{1-\beta}{\alpha}$, and H_0 when $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ [72].

The idea behind the SPRT is that we do not have to force a decision based on given data or a fixed size sample set, but it gives us the option to continue sampling, until we find a conclusive answer. Hence, it provides us a stopping criterion that allows us to stop when we have obtained desired probabilities for the type I and II errors (i.e., the false rejection of a true null hypothesis and failing to reject a false null hypothesis). Note that in contrast to the Chernoff-Hoeffding bound, the SPRT is not restricted to Bernoulli random variables. It also works with discrete and continuous random variables.

Table 2.1: SPRT example execution.

i	x_i	$f(x_i,\theta_0)$	$f(x_i,\theta_1)$	$\log(f(x_i,\theta_1)/f(x_i,\theta_0))$	$\log\Lambda_m$
1	1	0.8	0.9	0.051	0.051
2	0	0.2	0.1	−0.301	−0.250
3	1	0.8	0.9	0.051	−0.199
4	0	0.2	0.1	−0.301	−0.500
5	1	0.8	0.9	0.051	−0.449
6	1	0.8	0.9	0.051	−0.397
7	1	0.8	0.9	0.051	−0.346
8	0	0.2	0.1	−0.301	−0.647
9	1	0.8	0.9	0.051	−0.596
10	1	0.8	0.9	0.051	−0.545
11	0	0.2	0.1	−0.301	−0.846
12	1	0.8	0.9	0.051	−0.795
13	0	0.2	0.1	−0.301	−1.096
14	1	0.8	0.9	0.051	−1.045
15	0	0.2	0.1	−0.301	−1.346
16	1	0.8	0.9	0.051	−1.295
17	0	0.2	0.1	−0.301	−1.596
18	1	0.8	0.9	0.051	−1.545
19	0	0.2	0.1	−0.301	−1.846
20	1	0.8	0.9	0.051	−1.794
21	1	0.8	0.9	0.051	−1.743
22	0	0.2	0.1	−0.301	−2.044

Example. With the SPRT we can, e.g., check if the probability that a system can react to an input sequence (with a fixed size) without failure is closer to 0.9 or 0.8. Such a question can be expressed with the following hypotheses:

$$H_0 : \theta = \theta_0 \mid f(x_i,\theta_0) = \begin{cases} 0.2 & \text{if } x_i = 0 \\ 0.8 & \text{if } x_i = 1 \end{cases} \qquad H_1 : \theta = \theta_1 \mid f(x_i,\theta_1) = \begin{cases} 0.1 & \text{if } x_i = 0 \\ 0.9 & \text{if } x_i = 1 \end{cases}$$

Moreover, we need to define the bounds for the indifference region. For example, with $\alpha = 0.01$ and $\beta = 0.01$ the bounds are $\log \frac{\beta}{1-\alpha} = -2$ and $\log \frac{1-\beta}{\alpha} = 2$.

In order to perform such a test, we have to run the system with a generated input sequence to produce a sample, and calculate the log-likelihood ratio (2.3) according to the result.

An example SPRT execution is shown in Table 2.1. The first column indicates the current sample index, the second column represents the result of the sample ($x_i = 1$ means that there was no failure), the third and the fourth show the application of the probability density functions, the fifth shows the log-likelihood ratio, and the last column shows the cumulative sum of the ratio. It can be seen that it takes 22 samples for the algorithm to reach the lower bound of our indifference region, which means that H_0 was accepted, i.e., the probability was closer to 0.8.

2.2.4 Cumulative Sum (CUSUM)

CUSUM [114] is a sequential analysis technique similar to SPRT, because it also applies the log-likelihood ratio. However, in contrast to SPRT, its purpose is to detect a change of an initial probability. Given a finite set of independent Bernoulli random variables $X_1, \dots, X_n$, a probability for detecting a change $k \in [0,1]$ and sensitivity threshold λ, we want to decide between the hypotheses:

$H_0 : \forall i, 1 \leq i \leq N, p_i < k$ and
$H_1 : \exists i, 1 \leq i \leq N$ and $i : \forall n, 0 \leq n \leq N,$ such that $n < i \implies p_n < k$ and $n \geq i \implies p_n \geq k,$

where H_0 states that no change occurred and H_1 that a change occurred at time i, after which the probability is greater than k. We assume that we know the probability under normal conditions p_{init}, which can, e.g., be determined with a Monte Carlo simulation. We calculate the log likelihood-ratio s_i and the cumulative sum S_i as follows:

$$S_i = \sum_{i=1}^{n} s_i, \quad s_i = \begin{cases} \log(\frac{k}{p_{init}}) & \text{if } x_i = 1 \\ \log(\frac{1-k}{1-p_{init}}) & \text{otherwise} \end{cases} \tag{2.4}$$

We stop sampling when $S_n - \min_{1 \leq i \leq n}(S_n) \geq \lambda$, which means that a change $p_n \geq k$ was detected at time t_n or when no change occurred after a specified number of samples.

The CUSUM algorithm works, because of the fact that the cumulative sum (2.4) is globally decreasing when there is no change and continuously increasing after a change occurred. The algorithm stops if the increase is high enough, i.e., over the specified sensitivity threshold λ. Note, to find a good sensitivity threshold, it is necessary to perform several simulations and observe the maximum increases that are not caused by a change in order to avoid false positives due to local increases.

Example. With the CUSUM algorithm, we can, e.g., test if a change in the probability that a system has a failure can be observed. Moreover, if a change occurred, we can find when it was detected. In order to perform such a test, the initial failure probability without a change, needs to be known. We obtain it with a standard Monte Carlo simulation. For this example, we have an initial probability $p_{init} = 0.5$ and a change should be detected at probability $k = 0.7$ with an assumed sensitivity threshold of $\lambda = 2$. Table 2.2 demonstrates a run of the CUSUM algorithm. The first column is the sample index, the second column shows the outcome of the sample ($x_i = 1$ means there was a failure), the third illustrates the log likelihood-ratio, the forth the sum of the ratios, the fifth the minimum ratio and the last column shows the difference of the ratio and the minimum. It can be seen that up to the 20th sample, we have about the same number of successful and failing system runs. After that, there is a change, which causes an increase in the failure rate and also in the cumulative sum S_n. This increase continues up to the point, where $S_n - \min_{1 \leq i \leq n}(S_n) > \lambda$, which is the case at the 40th sample. At this point we can stop, because we found the change. Note that in a more realistic setting much higher values for the sensitivity threshold might be needed.

2.3 Linear Regression

For our model-based prediction method, we need response-time distributions in order to extend our functional model.

How can we derive such distributions? Implementing a classical rule-based algorithm is not feasible since appropriate *if-then-else* rules with the associated conditional expressions and calculation formulas for the distribution parameters are hard to define a priori in our context. However, we can recognize that we have all the necessary ingredients for a data-driven learning approach, more precisely, for supervised learning with regression [190]. We have log files with a large number of request examples (instances) for which also the response times (labels) are known. For each request example, the log file specifies the values of a

Table 2.2: CUSUM example execution.

i	x_i	s_i	S_n	$\min_{1 \le i \le n}(S_n)$	$S_n - \min_{1 \le i \le n}(S_n)$
1	1	0.146	0.146	0.146	0
2	0	−0.222	−0.076	−0.076	0
3	1	0.146	0.070	−0.076	0.146
4	0	−0.222	−0.151	−0.151	0
5	1	0.146	−0.005	−0.151	0.146
6	1	0.146	0.141	−0.151	0.292
7	1	0.146	0.287	−0.151	0.438
8	0	−0.222	0.065	−0.151	0.217
9	0	−0.222	−0.157	−0.157	0
10	1	0.146	−0.011	−0.157	0.146
11	1	0.146	0.136	−0.157	0.292
12	1	0.146	0.282	−0.157	0.438
13	0	−0.222	0.060	−0.157	0.217
14	0	−0.222	−0.162	−0.162	0
15	1	0.146	−0.016	−0.162	0.146
16	0	−0.222	−0.238	−0.238	0
17	0	−0.222	−0.460	−0.460	0
18	0	−0.222	−0.681	−0.681	0
19	0	−0.222	−0.903	−0.903	0
20	0	−0.222	−1.125	−1.125	0
21	0	−0.222	−1.347	−1.347	0
22	1	0.146	−1.201	−1.347	0.146
23	1	0.146	−1.055	−1.347	0.292
24	1	0.146	−0.909	−1.347	0.438
25	1	0.146	−0.763	−1.347	0.585
26	1	0.146	−0.616	−1.347	0.731
27	0	−0.222	−0.838	−1.347	0.509
28	1	0.146	−0.692	−1.347	0.655
29	0	−0.222	−0.914	−1.347	0.433
30	1	0.146	−0.768	−1.347	0.579
31	1	0.146	−0.622	−1.347	0.725
32	1	0.146	−0.476	−1.347	0.871
33	1	0.146	−0.329	−1.347	1.018
34	1	0.146	−0.183	−1.347	1.164
35	1	0.146	−0.037	−1.347	1.310
36	1	0.146	0.109	−1.347	1.456
37	1	0.146	0.255	−1.347	1.602
38	1	0.146	0.401	−1.347	1.748
39	1	0.146	0.547	−1.347	1.894
40	1	0.146	0.693	−1.347	2.040

number of attributes (features) related to the requests. Our regression task is to learn from the (labelled) data in the log files, a function which, given the attribute values of a request instance, returns the parameters (μ, σ) of a normal distribution for the response time for that instance. An analysis of the probability density functions of specific request types has shown that a normal distribution fits well enough.

As we will see in Chapter 5, it turns out that the response times can be fairly well approximated by a linear combination of the request attributes by using the linear regression method. This is convenient since (i) the statistical properties of the resulting estimators, i.e., the weights of the request attributes, are easier to determine with linear regression than with other learning algorithms, and (ii) we can use these statistical properties to derive the normal distribution parameters of the response times.

Multiple Linear Regression. The general linear regression model in matrix notation is defined as

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\epsilon} \tag{2.5}$$

where $\boldsymbol{y}$ is the dependent variable (regressand), $\boldsymbol{X}$ is the design matrix of the independent or explanatory variables (regressors), $\boldsymbol{\beta}$ contains the model parameters (regressor coefficients or weights), and $\boldsymbol{\epsilon}$ is the error term (noise) which captures all other factors which influence the dependent variable other than the regressors [78]. In more detail, in case of p regressors the i^{th} observation of the dependent variable is given by

$$y_i = 1\beta_0 + X_{i,1}\beta_1 + ... + X_{i,p}\beta_p + \epsilon_i \tag{2.6}$$

with β_0 as the constant or offset term (intercept). The case with more than one independent variable is called multiple linear regression (MLR). Thus, we use MLR to model the relationship between the response time, i.e., the dependent variable, and the attributes, i.e., the independent variables, of a request.

Given a log file with N examples of requests and their response times, $\boldsymbol{y}$ is the $N \times 1$ vector of the response times and $\boldsymbol{X}$ is the $N \times p$ design matrix for p request attributes considered to linearly influence the response time, where y_i is the response time and $X_{i,1}$, ..., $X_{i,p}$ are the attributes of the i^{th} request example in the log file.

We can use $\boldsymbol{y}$ and $\boldsymbol{X}$ with the equation (2.5) to estimate the model parameters $\boldsymbol{\beta}$ that minimise the error term $\boldsymbol{\epsilon}$. Note that $\boldsymbol{\epsilon}$ is a $N \times 1$ vector and there are various ways to define what "minimise $\boldsymbol{\epsilon}$" means. The simplest and most common method is the ordinary least squares (OLS) which minimises the sum of the squares ϵ_i^2, $i = 1, \ldots, N$.

After we have estimated the parameters $\boldsymbol{\beta} = [\beta_0, \beta_1, \ldots, \beta_p]$, we use the formula

$$y = 1\beta_0 + x_1\beta_1 + \ldots + x_p\beta_p \tag{2.7}$$

to predict the response time y of a new (unseen) request with attributes $[x_1, x_2, \ldots, x_p]$. Please note that the formula (2.7) is similar to (2.6) but without a correction error term ϵ which accounts for random variation or other unknown factors. Hence, (2.7) is an approximation of the real response time which can be evaluated by analysing the statistics (e.g., standard error, p-value, confidence interval) of the estimated model parameters $\boldsymbol{\beta}$ computed when applying the OLS method.

If we consider that the model parameters β_k, $k = 0, \ldots, p$, are normally distributed with the mean and standard deviation estimates $(\mu_{\beta_k}, \sigma_{\beta_k})$ given by the model parameters and the corresponding standard errors computed with OLS, then it follows that the predicted response time y is normally distributed with the mean μ_y and standard deviation σ_y given by

$$\mu_y = \sum_{k=0}^{p} x_k \mu_{\beta_k}, \qquad \sigma_y^2 = \sum_{k=0}^{p} x_k^2 \sigma_{\beta_k}^2 \tag{2.8}$$

as a linear combination of the normal distributions $\mathcal{N}(\mu_{\beta_k}, \sigma_{\beta_k}^2)$ with weights x_i, $i = 0, ..., p$ and $x_0 = 1$, according to (2.7). Note, this is based on the fact that a linear combination of normal distributions gives us a normal distribution as well.

The normal distribution $\mathcal{N}(\mu_y, \sigma_y^2)$ with parameters given by (2.8) is exactly what we are looking for. Thus, given a log file of request examples with corresponding response times, we learn the parameters $(\mu_{\beta_k}, \sigma_{\beta_k})$ of a model (2.8) which gives the normal distribution $\mathcal{N}(\mu_y, \sigma_y^2)$ of the response time y for any new request with known attributes $[x_1, ..., x_p]$ to be associated to the behavioural model as needed.

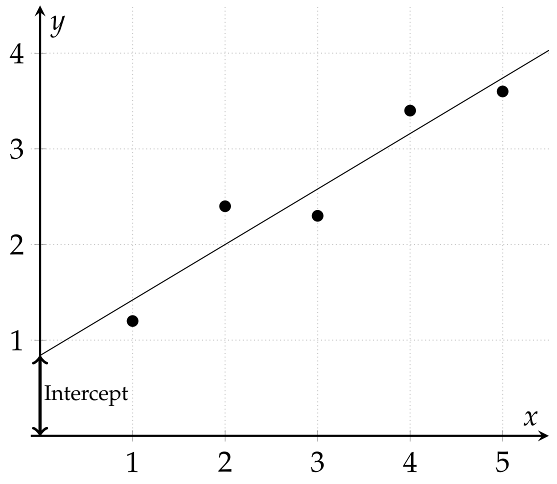


Figure 2.1: Linear regression example for two-dimensional data points.

Example. In the case of simple two-dimensional data points, we just need to calculate the slope (β_1) and the intercept (β_0) in order to compute a regression line as illustrated in Figure 2.1. Given the means of the data points $(\overline{x}, \overline{y})$, the slope and the intercept can be calculated based on the data points (x_i, y_i) of the figure as follows:

$$\mu_{\beta_1} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} = 0.58, \quad \mu_{\beta_0} = \overline{y} - \mu_{\beta_1}\overline{x} = 0.84$$

By taking $\mu_{\beta_0}, \mu_{\beta_1}$ as estimates for β_0, β_1 we obtain the regression line $y = 0.84 + 0.58x$, as illustrated in Figure 2.1, according to equation (2.7). Based on the predicted values $\hat{y}$ of the regression line and the residual standard error $\hat{\sigma}$, the standard errors of the slope σ_{β_1} and of the intercept σ_{β_0} can be computed by:

$$\sigma_{\beta_1} = \sqrt{\frac{\hat{\sigma}^2}{\sum_{i=1}^{n}(x_i - \overline{x})^2}} = \sqrt{\frac{\frac{\sum_{i=1}^{n}(y_i - \hat{y})^2}{n-2}}{\sum_{i=1}^{n}(x_i - \overline{x})^2}} = 0.11, \quad \sigma_{\beta_0} = \sqrt{\frac{\hat{\sigma}^2 \sum_{i=1}^{n} x_i^2}{n\sum_{i=1}^{n}(x_i - \overline{x})^2}} = 0.37$$

In order to obtain the normal distribution for a specific value, e.g., $x = 2$, we can just need formula (2.8) to calculate $\mu_y = 1\sigma_{\beta_0} + 2\sigma_{\beta_1} = 2$ and $\sigma_y = 1\sigma_{\beta_0}^2 + 2^2\sigma_{\beta_0}^2 = 0.19$ which gives us $\mathcal{N}(2, 0.19)$ that we can apply to obtain a sample.

2.4 Stochastic Timed Automata

Timed automata (TA) were originally introduced by Alur and Dill [15]. Here, we adopt the definition of UPPAAL [31]. A timed automaton (TA) is a tuple (L, l_0, A, C, I, E), where L is a finite set of locations, $l_0 \in L$ is the initial location, A is a finite set of actions, C is a finite set of non-negative real-value clocks, $I : L \to \mathcal{B}(C)$ is a finite set of invariants for the locations, where $\mathcal{B}(C)$ is a set of conditions for the clocks, and $E \subseteq L \times A \times \mathcal{B}(C) \times 2^C \times L$ is a finite set of edges between locations, with an action, a guard and a set of clock resets. Edges can also be written as $l \xrightarrow{a,g,r} l'$.

Given a location $l \in L$ and clock valuation $u : C \to \mathbb{R}_{\geq 0}$ for a TA state (l, u) and the set of all clock valuations $\mathbb{R}^C$, the semantics of a TA is defined by a labelled transition system $(S, s_0, \to)$. $S \subseteq L \times \mathbb{R}^C$ is a finite set of states, $s_0 = (l_0, u_0)$ is the initial state with $u_0 = \mathbf{0}$, where $\mathbf{0}$ means that all clocks $c \in C$ are set to zero. The transition relation is: $\to \subseteq S \times \{\mathbb{R}_{\geq 0} \cup A\} \times S$, and we have the following transitions:

- delayed transitions: $(l, u) \xrightarrow{d} (l, u + d)$ if $\forall d' : 0 < d' \leq d \implies u + d' \in I(l)$
- discrete transitions: $(l, u) \xrightarrow{a} (l', u')$ if $\exists (l \xrightarrow{a,g,r} l') : u \models g$, $u' = [r \mapsto 0]u$ and $u' \in I(l)$

Each clock $c \in C$ obtains the value $u(c) + d$ when $u + d$ is applied, $u \models g$ means that the guard g is true for the clock valuation u, and a clock reset $[r \mapsto 0]u$ sets all clocks in r to zero. It can be seen that delayed transitions let time pass, which allow us to represent the sojourn time, i.e., the delay that can occur in the locations of a TA. Discrete transitions take no time, but they may reset clocks. Note that we lift the plus operator to the clock valuation as follows: $u + d =_{def} \{c \mapsto u(c) + d \mid c \in C\}$.

Several probabilistic extensions of timed automata [15] have been proposed including stochastically enhanced TA [35] and continuous probabilistic TA [107]. Here, we follow the definition of stochastic timed automata (STA) by Ballarini et al. [25]: an STA is a tuple $(L, l_0, A, C, I, E, F, W)$ comprising a classical timed automaton (L, l_0, A, C, I, E), probability density functions $F = (f_l)_{l \in L}$ for the sojourn time, and natural weights $W = (w_e)_{e \in E}$ for the edges.

The transition relation can be described as follows. For a state given by the pair (l, u), the probability density function f_l is used to choose the sojourn time d, which changes the state to $(l, u + d)$. After this change, an edge e is selected out of the set of edges $E(l, u + d)$ that are enabled in a state $(l, u + d)$ with probability $w_e / \sum_{h \in E(l,u+d)} w_h$. Then, a transition to the target location l' of e and $u' = u + d$ is performed. For our models the underlying stochastic process is a semi-Markov process [92], since we reset our clocks at every transition, but we do not assume exponentially distributed waiting times, and therefore, the process is not a standard continuous-time Markov chain [25, 44, 195].

Example. An example stochastic timed automaton of a stochastic faulty counter is illustrated in Figure 2.2. This counter has increment *Inc* and decrement *Dec* transitions that normally change the value, i.e., the state, by one, but the increment does not always work. It may also do nothing, which happens in one percent of the cases. This faulty behaviour is represented by the two branches that are possible after an *Inc* transition: one branch (*IncF*) leads back to previous state with a weight of one and the other branch (*IncS*) leads to the next (correct) state with a weight of 99.

Moreover, we have additional states that represent delays for an increment and a decrement. For the increments, these additional states (I_i) apply a uniform distribution given by an upper and lower bound $[a, b]$ and for the decrements, we have additional states (D_i) with a normal distribution, given by the mean μ and the standard deviation σ. It can be seen that the parameters of the uniform and normal distributions are increasing, when the counter value becomes higher. This behaviour represents that the counter slows down, when its value increases.

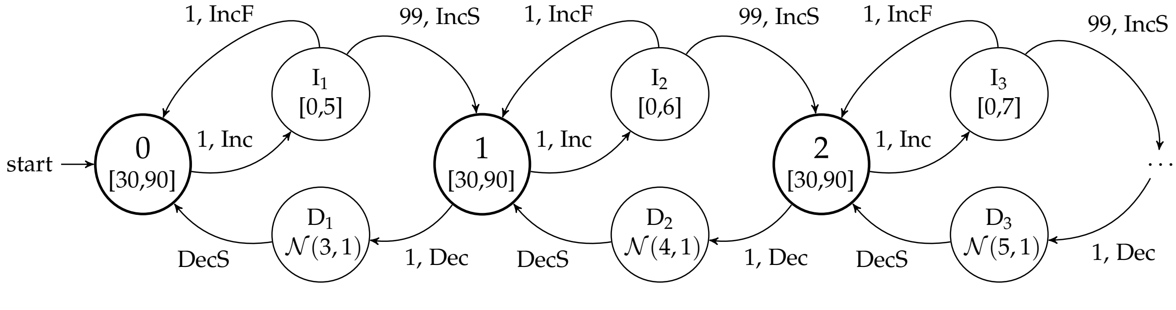


Figure 2.2: Stochastic timed automaton of a faulty slow counter.

Note that the *Inc*, *Dec* transitions also have weights, which represent the usage behaviour of the counter. In this example, both *Inc*, *Dec* have weight one, hence they are selected with the same probability. Moreover, we have uniform distributions $[30, 90]$ in the main counter states, which should illustrate the input time that is needed by the user. Later, we will see how we combine our stochastic models with more advanced usage profiles.

A run of such a model or of STA in general can be defined as: $(l_0, u_0) \xrightarrow{d_1, a_1} (l_1, u_1) \xrightarrow{d_2, a_2} \dots$ and it produces a timed trace in the form $(d_1, a_1), (d_2, a_2), \dots$, where d_i is a delay and $a_i \in A$. An example trace for this stochastic counter is the following:

```
(38.4,Inc),(2.3,IncS),(78.1,Inc),(3.7,IncS),(46.0,Dec),(3.4,DecS),
(73.4,Inc),(1.1,IncF),(67.2,Dec),(3.9,DecS),(34.9,Inc),(4.1,IncS)
```

3 Property-Based Testing with Business-Rule Models

This chapter is based on our publication in the journal SoSyM 2017 [5], at A-MOST 2016 [4] and at A-MOST 2017 [10].

3.1 Overview

Property-based testing is well suited for web-service applications, which was already shown in various case studies [64, 109]. For example, it has been demonstrated that JavaScript Object Notation (JSON) schemas can be used to automatically derive test-case generators for web forms. In this chapter, we present a test-case generation approach for a rule engine driven web-service application. Business-rule models serve as input for property-based testing. We parse these models to automatically derive generators for sequences of web-service requests together with their required form data.

In the past, PBT was mostly applied in the context of functional programming. Here, we define our properties in an object-oriented style in C# and its tool FsCheck. We apply our method to the business-rule models of an industrial web-service application, i.e., the TFMS that was introduced in Section 1.4.2.

Many web services store configurations in XML files. Some web services also store workflow details and user-access rules in XML business-rule models [155, 156]. These XML definitions can be seen as an abstract specification of the service behaviour, which may serve as a basis to verify whether the service complies to this specified behaviour [128]. We present an automated approach that uses these business-rule models to derive FsCheck[8] models and generators that are applied to generate command sequences with random input data.

The process of our testing approach is illustrated in Figure 3.1. The first step is to parse and translate the XML business-rule files to input models for FsCheck. FsCheck supports all kinds of models that have states, transitions, postconditions and optionally preconditions, but in our case the models were extended finite state machines (EFSMs) [45]. These EFSMs are used by the *specification builder* to create generators and FsCheck interface implementations according to the parsed model. FsCheck transforms these interface implementations into a property to be tested via randomly generated command sequences. This property requires that the state of the model is equal to the state of the SUT after each transition (command). As soon as a command sequence has been generated, it is executed on the SUT as a test case. When the property holds throughout the execution, then the test case was successful resulting in a *pass*-verdict, otherwise a *fail*-verdict is produced and the test case serves as a counterexample. The number of test cases can be specified by the user, but if a property fails, then no further test cases are executed.

For our use case a transition is not a simple action. It represents the opening of a page of a graphical user interface, the entering of data for form fields and saving the page. In the test-case generator, the transitions are realised as command classes with attributes representing the associated form data. Our target is to test the underlying requests of the transitions, which are necessary for the interaction with the web-service application.

The contributions in this chapter are the following:

1. The main contribution is a new testing approach that uses XML business-rule models in the form of EFSMs as input for PBT.

[8]`https://fscheck.github.io/FsCheck` (visited on 2018-09-19)

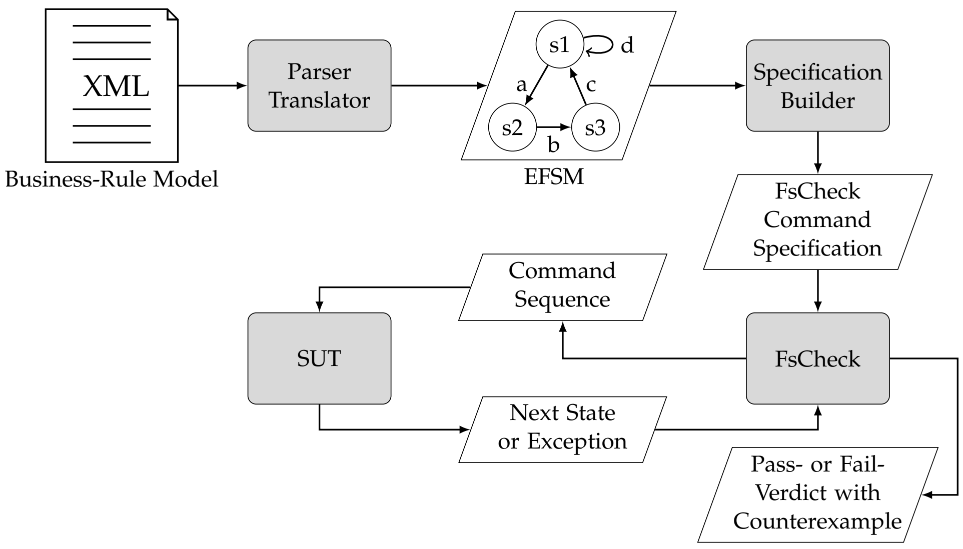


Figure 3.1: Overview of the steps for the test-case generation with business-rule models.

2. We formalise the underlying concepts and algorithms of PBT with EFSMs and present a detailed definition of our rule-engine models with an abstract grammar.
3. Moreover, we show a formalisation of the translation of our business-rule models to EFSMs.
4. Another contribution is the application and evaluation of our approach in an industrial case study.

The rest of the chapter is structured as follows. First, in Section 3.2, we present our rule-engine system and rule-engine models. In Section 3.3, we introduce PBT with EFSMs, and we present a small example of model-based testing with FsCheck. Section 3.4 presents application-specific extensions to our method, like the translation of business-rule models to EFSMs. Then, in Section 3.5 we describe details about the structure and implementation of our approach. Section 3.6 shows the results of an evaluation, where we performed two case studies for two major modules of an industrial web application. Moreover, we present an extension for PBT that exploits an external test-case generator in Section 3.7. Finally, in Section 3.8, we discus the limitations, threats to validity and future work, and conclude in Section 3.9.

3.2 Business-Rule Models

An application may need various modifications depending on the customer or on the country of deployment. It is infeasible to apply these modifications to the source code, because it would require the development of different versions for each customer. A business-rule engine is a good way to apply the different modifications in the form of rules for different deployments of an application. Business-rule engines are used to integrate these rules in the business logic. They are often combined with business-rule management systems that can be used to store, load and easily modify the rules. There are many frameworks, architectures or systems for web services and applications in general that provide business-rule management functionality [80, 140, 157].

Most of them only differ by the information that can be encoded in the rules. For example, business-rule engines can store constraints, conditions, actions and other business-process semantics. Even workflow details can be included, although there is a separate technology, called workflow engine or also business process management system [1, 43, 155]. The major difference is that workflows/processes define the order or sequence of tasks (actions/operations) and business rules describe conditions and resulting actions.

Our SUT (i.e., the TFMS) has a custom implementation of a rule management system. This custom implementation was made, because there were not many existing approaches at the time, when the application was developed. Our business-rule models are similar to other rule definitions. For example, the rule markup language (RuleML) [186] could be applied to encode our models.

Note that we talk about rules and not processes as our models have the main purpose of storing costumer specific business logic, and they specify conditions for enabling certain tasks, which can be seen as condition-action pairs. They do not focus on the sequence of tasks and do not support the composition of services. Moreover, the term rule-engine is used within the given commercial system.

In this work business-rule, models are also called rule-engine models (REMs) and they are the primary basis of our approach. An REM is a state machine defining the behaviour of a *TFMS Object Class*. A *TFMS Object Class* describes objects of our application domain, like incidents or test orders. Each of these objects has a state, an identifier, attribute values/data and they are stored in the database of our SUT.

The abstract syntax of a rule-engine model can be defined as follows. Its definition corresponds to the concrete XML syntax, but is more concise.

Definition 1 (Rule-Engine Models)**.**

$$\begin{aligned}
\mathit{REM} &=_{df} \mathit{rem}(\mathit{AllAttributes}, \mathit{AllTasks}, \mathit{AllStates})\\
\mathit{AllAttributes} &=_{df} \mathit{Attr}*\\
\mathit{Attr} &=_{df} \mathit{attr}(\mathit{Name}, \mathit{DataType}, \mathit{Parameter}*) \mid \mathit{attr}(\mathit{Name}, \mathit{DataType})\\
\mathit{DataType} &=_{df} \mathit{Integer} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Enum} \mid \mathit{Object} \mid \mathit{Date} \mid \mathit{DateTime} \mid \mathit{Float} \mid \mathit{File} \mid \mathit{Reference}\\
\mathit{Parameter} &=_{df} \mathit{MinValue} \mid \mathit{MaxValue} \mid \mathit{EnumItem}* \mid \mathit{Query} \mid \mathit{Regex} \mid \ldots\\
\mathit{AllTasks} &=_{df} \mathit{Task}*\\
\mathit{Task} &=_{df} \mathit{task}(\mathit{id} : \mathit{Name},\ \mathit{attributes} : \mathit{Name}*, \mathit{possibleNextStates} : \mathit{Name}*)\\
\mathit{AllStates} &=_{df} \mathit{State}*\\
\mathit{State} &=_{df} \mathit{state}(\mathit{id} : \mathit{Name},\ \mathit{possibleTasks} : \mathit{Name}*)
\end{aligned}$$

For easier readability we use *record types* to define composite data: an REM is defined as a record *rem* with three fields: the set of *AllAttributes*, the set of *AllTasks* and the set of *AllStates*. In fact, these sets are represented as sequences, e.g., the sequence of all attributes *Attr*∗. An attribute comprises a *Name*, a *DataType* and optionally a sequence of *Parameter*s. Parameters may further restrict a data type, like a maximum value for an *Integer*. A more complex form of restriction of an attribute may be realised via a *Query* to a database, which will be further explained in Section 3.5.1 (reference attributes). This allows to implement a selection of existing values, like e.g., a dynamic drop-down menu in a web-form. Another restriction can be applied with a regular expression (*Regex*), which can limit a string attribute to only allow certain patterns.

Tasks represent the behaviour, i.e., the actions or events a user may trigger. For readability, we define tasks with *field names*. A *task* has an identifier, i.e., a *Name*, a number of attributes to be entered into a form and the possible next states a task may reach. If there is more than one possible state, then it can be selected via an external choice by the user, hence this does not represent non-determinism. Finally, all *States* define the complete state-space with each state being associated with a sequence of tasks that can be triggered in this state.

For illustration, Listing 3.1 shows a simplified version of the XML file of an REM that was used as basis for the example in Section 3.3.2. It can be seen that these models are structured very similarly to our abstract syntax. The main components are:

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <RuleEngineModel TfmsType="Incident">
 3   <AllAttributes>
 4     <StaticAttributeInfo Name="ParentFolder" DataType="Reference">
 5       <Query Criteria="Class=IncidentFolder">
 6         <RequestedAttributes> <string>*</string> </RequestedAttributes>
 7       </Query>
 8     </StaticAttributeInfo>
 9     <StaticAttributeInfo Name="Description" DataType="String" MaxValue="128" />
10     ...
11   </AllAttributes>
12   <AllTasks>
13     <Task Name="IncidentCreateTask">
14       <DynamicAttributesInfo>
15         <Attribute Name="ParentFolder" Enabled="true" Required="true" />
16         <Attribute Name="Description" Enabled="true" Required="true" />
17         ...
18       </DynamicAttributesInfo>
19       <PossibleNextStates>
20         <State Name="Submitted" NoteRequired="false" />
21       </PossibleNextStates>
22     </Task>
23     ...
24   </AllTasks>
25   <AllStates>
26     <State Name="Submitted">
27       <PossibleTasks>
28         <Task>IncidentEditTask</Task>
29         <Task>IncidentCloseTask</Task>
30       </PossibleTasks>
31     </State>
32     ...
33   </AllStates>
34 </RuleEngineModel>
```

Listing 3.1: Simplified XML representation of a rule-engine model.

- attribute definitions with data types and constraints (Lines 3 to 11)
- tasks with enabled and required attributes and possible next states (Lines 12 to 24)
- states with possible tasks (Lines 25 to 33)

Optionally the models may also include:

- scripts, which can be executed on certain events
- queries for the selections of specific objects
- reports for a good overview of the entered objects

Note that our REMs do not always represent the actual behaviour of the web application under test. REMs determine what tasks are currently active and what attributes are required. However, developers can overrule the constraints that are included in REMs, when they implement a task. For example, a task can lead to different target states that are not specified in an REM. Moreover, a form of the SUT might require additional attributes for special cases that are only implemented in the SUT, but not contained in REMs. It makes sense to search for such cases where REMs are overruled by the implementation in order to find out, if this behaviour is intentional or was introduced by mistake. In addition, manual adjustments to the test models had to be made so that these deviations are not found repeatedly.

3.3 Property-Based Testing with Extended Finite State Machines

As already explained in Section 2.1, PBT can be applied for various types of models. Now, we introduce PBT with extended finite state machines. We formalise the underlying concepts and algorithms in the first subsection and present concrete examples in the second subsection.

3.3.1 State-Machine Properties

PBT can be applied to models in the form of extended finite state machines (EFSMs) [97].

Definition 2 (EFSM)**.** An EFSM can formally be defined as a 6-tuple $(S, s_0, V, I, O, T) \in State\_set \times State \times Variable\_set \times Input\_set \times Output\_set \times Transition\_set$, where
S is a finite set of *State*s,
$s_0 \in S$ is an initial *State*,
V is a finite set of *Variable*s,
I is a finite set of *Input*s,
O is a finite set of *Output*s,
T is a finite set of transitions, $t \in T$ can be defined as a 5-tuple (s, i, g, op, s'),
s is the source *State*,
i is an *Input*,
g is a guard in the form of a Boolean expression,
op is a sequence of output and assignment operations of the form $output := o$ or $v := e$, where $output$ is a keyword, $o \in O$, $v \in V$ and e is an expression,
s' is the target *State* [97].

Such an EFSM is deterministic if in any state there is at most one enabled transition (via guards) with the same input [166]. In this thesis, we are concerned with deterministic EFSMs.

An example EFSM is presented in Section 3.3.2 in Figure 3.3. Semantically, a guard g is a Boolean function that takes the variable valuations v as input and returns a Boolean value. An operation op is a function mapping the current variable valuations to a pair of new valuations and an optional output $o \in O$.

In order to perform PBT for an EFSM, a state-machine specification *spec* has to be provided. This *spec*ification includes functions to set the initial state of the model and the SUT, a set of commands *cmds* and a *next* function that builds a command generator *Gen*[*Cmd*] for a given model state:

Definition 3 (State-Machine Specification)**.**

$$\begin{aligned} Spec \;=_{df}\; & spec(initalModel : () \rightarrow Model,\, initialActual : () \rightarrow Sut, \\ & cmds : Cmd\_set,\; next : Model \rightarrow Gen[Cmd]) \end{aligned}$$

Algorithm 3 in Section 3.3.2 outlines an example specification for the incident manager as it is required for FsCheck.

A *Model* object consists of fields representing the current EFSM state s, the valuations for the variables v, the transition set T, the last output o and a *doStep* function that performs the execution of a transition.

$$\begin{aligned} Model \;=_{df}\; & model(s : State, v : Variable \rightarrow Val, T : \\ & \quad Transition\_set, o : Output, doStep : Input \rightarrow Model) \\ doStep(in) \;=_{df}\; & model(s', v', o', doStep) \text{ such that} \\ & \quad (s, in, g, op, s') \in T \wedge g(v) = True \wedge (v', o') = op(v) \end{aligned}$$

Note that the SUT is defined in the same way as the model and is, therefore, omitted.

A command $Cmd_{in} \in spec.cmds$ encodes a set of transitions T_{in} with the same input *in*. They encapsulate preconditions, postconditions and the execution semantics of these transitions. Preconditions *pre* define the permitted transition sequences by enabling the command only in states where the input *in* is allowed. Postcondition *post* can verify the effects of the command, e.g., by comparing the state of the model and the SUT. The execution semantics are

$$
\begin{aligned}
Cmd_{in} &=_{df} cmd_{in}(T_{in} : Transition\_set, pre_{in} : Model \rightarrow Bool,\\
&\qquad post_{in} : (Model, Sut) \rightarrow Boolean,\\
&\qquad runModel_{in} : Model \rightarrow Model, runActual_{in} : Sut \rightarrow Sut)\\
T_{in} &=_{df} \{(s,i,g,op,s') \mid (s,i,g,op,s') \in T \wedge i = in\}\\
pre_{in}(model) &=_{df} \begin{cases} True & \text{if } \exists (s,in,g,op,s') \in T_{in}.\ s = model.s \wedge g(model.v) = True \\ False & \text{otherwise} \end{cases}\\
runModel_{in}(model) &=_{df} model.doStep(in)\\
runActual_{in}(sut) &=_{df} sut.doStep(in)\\
post_{in}(model, sut) &=_{df} \begin{cases} True & \text{if } model.s = sut.s \wedge model.v = sut.v \wedge model.o = sut.o \\ False & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.2: Command definition for property-based testing.

encoded via the functions *runModel* and *runActual* for executing the *Model* and the SUT. The definition of a command is shown in Figure 3.2. Note that in this definition we show various possible checks in the postcondition, i.e., we analyse the current state of the SUT, variable valuations and the output. In reality this may not be feasible, because the SUT might not provide all this information. Hence, in many cases it may only be possible to check the output in the postcondition. An example implementation of a command is presented in Section 3.3.2 in Algorithm 4. This example demonstrates the function definitions for an *IncidentCreateTask.*

A property of an EFSM is that for each command sequence that is possible with the preconditions, the postcondition of each command must hold. In order to verify this property, a PBT tool produces random command sequences and checks the postconditions after each command execution.

The state-machine property must hold in all settings of the model *Model_set* and the SUT *SUT_set* that are reachable via valid command sequences. A command sequence is valid if all its preconditions are satisfied. Hence, given a specification *spec*, a state-machine property for EFSMs can be defined as follows:

Definition 4 (State-Machine Property)**.**

$$
\begin{aligned}
&\forall cmd_{in} \in spec.cmds, model \in Model\_set, sut \in SUT\_set :\\
&cmd_{in}.post(model, sut) \wedge cmd_{in}.pre(model) \implies\\
&\qquad cmd_{in}.post(cmd_{in}.runModel(model), cmd_{in}.runActual(sut))
\end{aligned}
$$

Algorithm 1 shows the pseudo code of the test-case generation for such a property. The algorithm takes a *spec* and a *size* parameter for the length of the test case as input and returns a *testSequence*, which is a sequence of $(Cmd, Model)$ pairs. In the first step, the initial model is created with the *initialModel* function of the *spec*. Next, there is a loop over the size parameter. In each iteration, a command generator *gen* is built with the *next* function of the *spec*. This function takes the model (Line 9) and creates a subset of all commands by checking their precondition. The function returns an *Elements* generator, which selects one element of this set with a uniform distribution (Line 11). The sample function of this generator is called to produce a command (Line 4). This command is executed with *runModel*, which returns a new model that incorporates the state change. Note that we need a new model and not only change the current one, because future changes should not affect the old stored model instances. This new model and the command are stored in the *testSequence*. Finally, after the loop is finished we return the *testSequence*, which represents a test case.

Algorithm 1 Pseudo code of the test-case generation for EFSMs.

```
Input:
    $spec$: state-machine specification
    $size \in \mathbb{N}_{>0}$:parameter for test-case length
Output:
    $testSequence : (cmd_1, model_1), \ldots, (cmd_n, model_n)$
 1: $model \leftarrow spec.initialModel()$
 2: for $i \leftarrow 1$ to $size$ do
 3:     $gen \leftarrow spec.next(model)$                ▷ next returns a $cmd$ generator
 4:     $cmd \leftarrow gen.sample()$                    ▷ command is generated
 5:     $model \leftarrow cmd.runModel(model)$           ▷ command is executed
 6:     $testSequence[i] \leftarrow (cmd, model)$       ▷ build test sequence
 7: end for
 8: return $testSequence$

 9: function spec.next($model$)
10:     $cmdSet \leftarrow \{cmd_{in} \in spec.cmds \mid cmd_{in}.pre(model) = True\}$
11:     return $Gen.Elements(cmdSet)$
12: end function
```

Algorithm 2 shows how such a generated test case can be executed. The test case is the input of this algorithm together with a *spec* and the result is a verdict. (Note that shrinking is omitted in this simplified algorithm.) In the first step, the initial SUT is built by the *initialActual* function of the *spec*. After that we loop over the *testSequence*. Next, the command is executed on the SUT with *runActual*, which results in a modified SUT (Line 3). The postcondition of the command is applied to compare the SUT with the stored model of the *testSequence*. If it is false, then the test failed. Otherwise, the execution continues and if the loop is finished, then the postconditions of all commands were satisfied and a pass-verdict is returned.

3.3.2 Example of Model-Based Testing with FsCheck

In this subsection, we show how FsCheck can be applied for MBT. A simple example of an incident manager taken from our industrial case study shall serve to demonstrate how the necessary interface implementations have to be realised.

FsCheck Modelling. In order to apply FsCheck for MBT, we need a specification class that implements an ICommandGenerator interface and contains the following elements:

- SUT definition (which is called Actual by FsCheck)

Algorithm 2 Pseudo code of the test-case execution for EFSMs.

```
Input:
    $testSequence : (cmd_1, model_1), \ldots, (cmd_n, model_n)$
    $spec$ : state-machine specification
Output:
    $Pass$, if the test case is successful, $Fail$ otherwise
 1: $sut \leftarrow spec.initialActual()$
 2: for each $(cmd_i, model_i) \in testSequence$ do
 3:     $sut \leftarrow cmd_i.runActual(sut)$            ▷ command is executed
 4:     if $\neg cmd_i.post(model_i, sut)$ then          ▷ check post condition
 5:         return $Fail$
 6:     end if
 7: end for
 8: return $Pass$
```

Algorithm 3 Incident specification *Spec* of the incident manager.

```
Input:
    SUT class for the connection to the SUT,
    Model class
 1: function initialActual
 2:     return new SUT()                                   ▷ create new SUT instance
 3: end function

 4: function initialModel
 5:     return new Model()
 6: end function

 7: cmds ← {new IncidentCreateTask(), new IncidentEditTask(),new IncidentCloseTask()}
 8: function next(model)
 9:     return Gen.Elements(cmds)                          ▷ chose one element
10: end function
```

- Model definition
- Initial state of the SUT and the model
- Generator for the next command given the current state of the model
- Commands combining preconditions, postconditions and the transition execution semantics of the SUT and the model

Details about the structure of such specifications were already presented in Section 3.3. Now we give a more concrete example for FsCheck on an object-oriented level. Algorithm 3 outlines an example specification. In order to implement the interface for FsCheck, we need the mentioned elements. The class of the SUT is basically a wrapper that provides methods for the execution of all tasks and a method to retrieve the current state of one incident object of the SUT.

An incident object is an element of the application domain. For example, it could be a bug report. It has a number of attributes (form data), which are stored in the database. In this example, we assume that the attributes are set statically in the wrapper class of the SUT. In Section 3.5, we will see, how form data can be generated automatically for these attributes.

Figure 3.3 illustrates the state machine of one incident object. Initially, the machine is in a global state. The IncidentCreateTask (abbreviated as Create) creates and opens a new incident object, which can be edited and closed with the corresponding tasks. The transitions are labelled as follows: input *i*, an optional guard *g* / assignment operations *op*, and an output *o*. The assignment operations of this EFSM assign values to the attribute variables and the output indicates the target state of a transition. The initial global state has a special meaning: tasks of

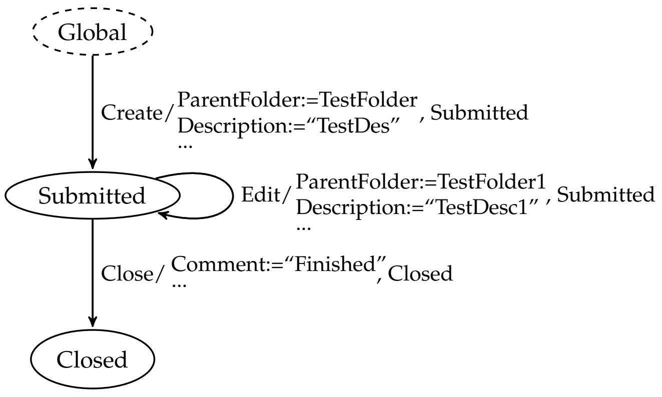

Global
Create/ ParentFolder:=TestFolder Description:="TestDes" ... , Submitted
Submitted
Edit/ ParentFolder:=TestFolder1 Description:="TestDesc1" ... , Submitted
Close/ Comment:="Finished" ... , Closed
Closed

Figure 3.3: EFSM of the Incident Manager.

Algorithm 4 IncidentCreateTask command definition of the incident manager.

```
function pre($model$)
    return $True$
end function

function post($sut, model$)
    return $sut.State = model.State$
end function

function runModel($model$)
    $model.doStep(\text{“IncidentCreateTask”})$
    return $model$
end function

function runActual($sut$)
    $sut.doStep(\text{“IncidentCreateTask”})$
    return $sut$
end function

function toString
    return “IncidentCreateTask”
end function
```

the global state, i.e., IncidentCreateTask, are globally enabled in all states. Hence, it is possible in every state to create new incident objects. However, to simplify the discussion, we assume that the state machine only represents a currently opened incident object. Generally, in an object-oriented system comprising several objects, we need functionality to switch between active objects. This functionality is discussed in Section 3.4.2.

The initial states of the model and the SUT are set by creating new objects (Algorithm 3: Lines 2 and 5). The generator in the *next* function selects one element of a command set randomly, which can be accomplished with the default *Elements* generator of FsCheck (Line 9).

In the standard PBT approach, all command classes need to be defined manually as shown in Algorithm 4. The classes need to define how the transitions should be executed on the model and SUT and what postcondition should hold after the execution. In this simple example, the execution of the model only changes the state, later we will also see how we handle form data. For example, the state-changing function of an IncidentCreateTask is defined as follows:

$$model.doStep(\text{“}IncidentCreateTask\text{”}) =_{df} \quad model.s := \text{“}Submitted\text{”}$$

Note that in contrast to the previous abstract definition, the postcondition here checks if the state of the SUT matches the state of the model. Moreover, a *toString* method can be used to display various information of the command and optionally a precondition can be defined. The classes for the IncidentEditTask and the IncidentCloseTask command are similar to this class and are, therefore, omitted.

For a large model with many transitions it is not practical that all commands have a separate class. Therefore, it makes sense to implement this definition in a more generic way for all possible transitions and to automate the process as far as possible.

Command Generation and Execution. The tool FsCheck generates test cases according to Algorithm 1 with the difference that the specification is provided in an object-oriented style as shown in Algorithm 3. After a test case is generated it is executed on the SUT and the state of the SUT is compared to the stored model state after each command execution as explained in Algorithm 2.

```
0:
[IncidentCreateTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCreateTask;IncidentEditTask]

1:
[IncidentCreateTask;IncidentEditTask;IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;IncidentCreateTask;IncidentEditTask;
IncidentCreateTask;IncidentCloseTask;IncidentCreateTask;IncidentEditTask;IncidentCreateTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;IncidentEditTask;IncidentCloseTask;
IncidentCreateTask;IncidentCreateTask;IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentEditTask;IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCreateTask;IncidentCreateTask;IncidentCloseTask;IncidentCreateTask;
IncidentCreateTask;IncidentEditTask;IncidentEditTask;IncidentCreateTask;IncidentEditTask;
IncidentCloseTask;IncidentCreateTask;IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentCreateTask;IncidentCloseTask;IncidentCreateTask;IncidentCreateTask]
Ok, passed 2 tests, 50% short sequences (1-6 commands)
```

Listing 3.2: Generated command sequences for the incident manager.

In order to start testing in FsCheck, the specification has to be converted into a property. This is achieved with the *toProperty*() method of FsCheck. The property can then, e.g., be tested by calling the QuickCheck() method or also with the help of unit testing frameworks:

```
new Spec().toProperty().QuickCheck();
```

By default, 100 test cases will be generated and executed, but this number can be configured. Listing 3.2 shows two example sequences that were produced by FsCheck for the incident specification. It can be seen that the sequences have quite different lengths, because FsCheck generates them randomly with a variety of lengths. Moreover, FsCheck classifies the sequences according to their lengths, which can be seen in the last line of the listing. These classifications can be helpful to find out that a certain generator only considers trivial cases. Each of these generated tasks in the command sequences requires form data for the attributes, which also needs to be generated. Listing 3.3 shows example form data for some attributes that was generated randomly for the IncidentCreateTask. Note that this randomly generated strings form a kind of robustness test in order to check that the SUT can process non-standard input.

3.4 Application-Specific Extensions to the Method

In this section, we present some application-specific extensions, which were needed for the web-service application that we evaluated. We show the translation of business-rule models into EFSMs, and we demonstrate how we introduced functionality to switch between application objects and REMs.

3.4.1 Translating Business-Rule Models into Extended Finite State Machines

In the following, we show how we translate our business-rule models into EFSMs. In order to do this, we introduce a function *translate* which takes an REM as described in Section 3.2 as

```
IncidentCreateTask:
-- ParentFolder = IncidentTestFolder1
-- Description = /%j6XN--#Gn8$-bc6_I-@EaB-cfkMNn3-->eA -sb!--R-9/{@bXzF--#o4*LB]SY3 --r{i-!-
    p-x-f
-- CommitNote = HC--Gz_p;
...
```

Listing 3.3: Generated form data for a task of the incident manager.

$$\begin{aligned}
&\mathit{translate} : \mathit{REM} \rightarrow \mathit{EFSM}\\
&\mathit{translate}(\mathit{rem}(\mathit{attributes}, \mathit{states}, \mathit{tasks})) =_{df} (\mathit{names\_of}(\mathit{states}), \text{“}\mathit{Global}\text{''},\\
&\quad \mathit{names\_of}(\mathit{attributes}), \mathit{names\_of}(\mathit{tasks}), \mathit{names\_of}(\mathit{states}), \mathit{buildTrans}(\mathit{states}, \mathit{tasks}, \mathit{attributes}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{buildTrans} : \mathit{AllStates\_set} \times \mathit{AllTasks\_set} \times \mathit{AllAttributes\_set} \rightarrow \mathit{Transition\_set}\\
&\mathit{buildTrans}(\mathit{states}, \mathit{tasks}, \mathit{attributes}) =_{df} \{(s, i, \mathit{true}, \mathit{op}, s') \mid \exists\, \mathit{state}(\mathit{sname}, \mathit{possibleTasks}) \in \mathit{states} \,.\, (s = \mathit{sname} \land\\
&\quad \exists\, \mathit{task}(\mathit{tname}, \mathit{tattributes}, \mathit{nextStates}) \in \mathit{tasks} \,.\, (\mathit{tname} \in \mathit{possibleTasks} \land s' \in \mathit{nextStates} \land\\
&\quad i = \mathit{tname} +\!+ \mathit{optionalSuffix}(\mathit{nextStates}, s') \land\\
&\quad \mathit{op} = (a_1 := v_1, \ldots, a_n := v_n, \mathit{output} := s') \land a_i \in \mathit{tattributes} \land (a_i, v_i) \in \mathit{translate}(\mathit{attributes})))\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{translate} : \mathit{AllAttributes\_set} \rightarrow (\mathit{Variable} \times \mathit{Generator})\_\mathit{set}\\
&\mathit{translate}(\mathit{attributes}) =_{df} \{(\mathit{id}, \mathit{gen}) \mid \mathit{attr}(\mathit{id}, \mathit{type}) \in \mathit{attributes} \land \mathit{gen} = \mathit{Gen}(\mathit{type}) \lor\\
&\qquad\qquad \mathit{attr}(\mathit{id}, \mathit{type}, \mathit{par}) \in \mathit{attributes} \land \mathit{gen} = \mathit{Gen}(\mathit{type}, \mathit{par})\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{names\_of} : \mathit{AllTasks\_set} \mid \mathit{AllStates\_set} \mid \mathit{AllAttributes\_set} \rightarrow \mathit{Name}\\
&\mathit{names\_of}(\mathit{xs}) =_{df} \{x.\mathit{name} \mid x \in \mathit{xs}\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{optionalSuffix} : \mathit{Name\_set} \times \mathit{Name} \rightarrow \mathit{Name}\\
&\mathit{optionalSuffix}(\mathit{possibleNextStates}, \mathit{nextState}) =_{df} \begin{cases} \text{“”} & \text{if } \mathit{card}(\mathit{possibleNextStates}) = 1\\ \mathit{nextState} & \text{if } \mathit{card}(\mathit{possibleNextStates}) > 1 \end{cases}
\end{aligned}$$

Figure 3.4: Translation of a rule-engine model to an EFSM by translating the attributes, tasks and states of the rule-engine model to the 6-tuple representing an EFSM.

input and converts it into an EFSM. Figure 3.4 illustrates this translation. It can be seen that the translate function is a composition of several sub-functions that convert the individual parts of the REM. First, we translate the states of the REM by applying the helper function *names_of*, which returns the name of all elements of a set of tasks, states or attributes. The name is a unique identifier for each element of these sets. The same helper function is used to translate the attributes, tasks and states to variable, input and output sets. The initial state of the EFSM is set to the constant "Global". The translation of the transitions is more complex and is performed with the *buildTrans* function. For this translation, we take the states, the tasks and the attributes of the REM as arguments, because we need their fields *possibleTasks* and *possibleNextStates* to form the transitions of the EFSM: the states of the REM form the source states, the tasks that are enabled in these states (*possibleTasks*) represent the inputs and their possible next states define the target states.

Note, there can be multiple possible next states for one task in a specific state. In practice, one of these states is selected by the user, which represents an additional external input. For example, in some REMs an *AdminEdit* task can be performed, where a user can select the next state of an object, like *Created*, *Available* or *Deleted*, explicitly from a drop-down menu. Since this selected state forms part of the input, we add the name of the state to the task name to form the input. The function *optionalSuffix* appends $(++)$ the next state s' to the input (task name), if more than one possible next state exists (*card*inality > 1). In the following, we skip this detail in order to simplify our graphical representations. Furthermore, we translate the required attributes of the tasks in a separate function, which returns a set of pairs of variables and generators (id, gen) for the types of the attributes and their optional parameters. These pairs and an output assignment form the operation sequences of a transition.

3.4.2 Switching Between Rule-Engine Models Objects

In this subsection, we explain how we extended our models to enable the switching between multiple application objects. The business-rule models only consider the behaviour

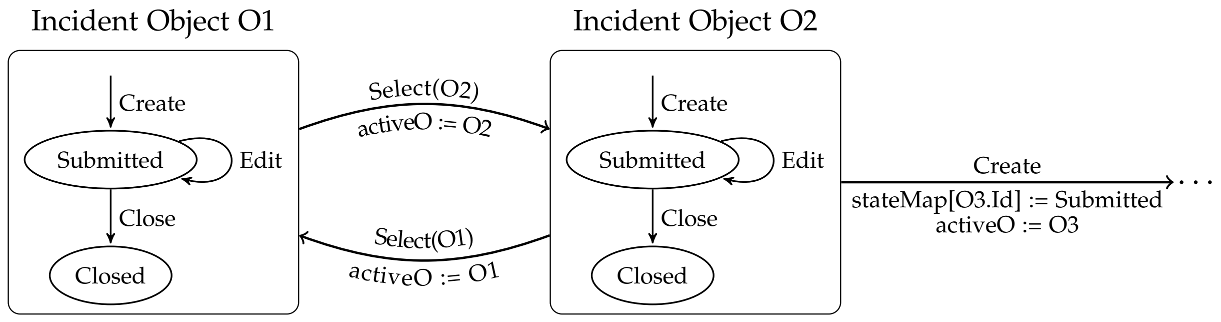


Figure 3.5: Switching between objects of the incident object class.

of one object of a specific type and not the behaviour of a set of objects. Our application includes a number of transitions that create new application objects and a user can switch between these objects before a task is started. Our original implementation only considered the currently active object, which is automatically changed to inactive when a new object is created. In order to also support the switch functionality, we extended our models. We changed the original variable set V of the EFSM to $V = V_{attr} \cup activeObj \cup stateMap$, where $V_{attr} : Obj.Id \rightarrow Variable\_set$ are the variables for our attributes. They are now represented in a map, so that different variables can be maintained for the different objects. The *activeObj* is a variable that marks the currently opened object and it contains an identifier and a state. The variable $stateMap : Obj.Id \rightarrow State$ was added to map object identifiers to object states in order to keep track of the current states of all objects. Moreover, we added additional transitions for selecting an object. These transitions are only enabled when at least two objects are available, and they are chosen randomly in the same way as other transitions. The decision, which object is selected, is also performed by a generator. Figure 3.5 illustrates the select functionality with the additional transitions, which is repented as a hierarchical state machine [77]. Note that transitions that create objects are always enabled, i.e., they are enabled globally in all states. In the following display of EFSMs, we skip this global initial state of the REMs.

In the same way as we added transitions to switch between objects of an REM, we also added transitions to switch between different REMs within a module. Figure 3.6 demonstrates, how a switch between multiple REMs of the Test Order Manager can be accomplished with SelectREM transitions. In this example, we have three EFSMs for the different REMs of this module, which are explained in more detail in Section 3.6.2. Each EFSM in this figure also has the select functionality to switch between objects of the REM as shown in Figure 3.5.

On top of the switching between REMs, we may also switch between modules as illustrated in Figure 3.7. In this figure, we have three modules: Test Order Manager, Test Equipment Manager and Test Factory Scheduler. The first two modules are both models we used for case studies in Section 3.6. The Test Factory Scheduler module is for scheduling of test orders on the test beds. It can automatically check the availability of resources like test beds and equipment. Test orders can be assigned according to priorities and to meet certain deadlines. We have not implemented the switch functionality at this level, because we wanted to test the modules separately, but it would be straightforward and very similar to the switch functionality of REMs within a module.

3.5 Architecture and Implementation

In this section, we discuss the architecture and the relevant implementation details of our test-case generator.

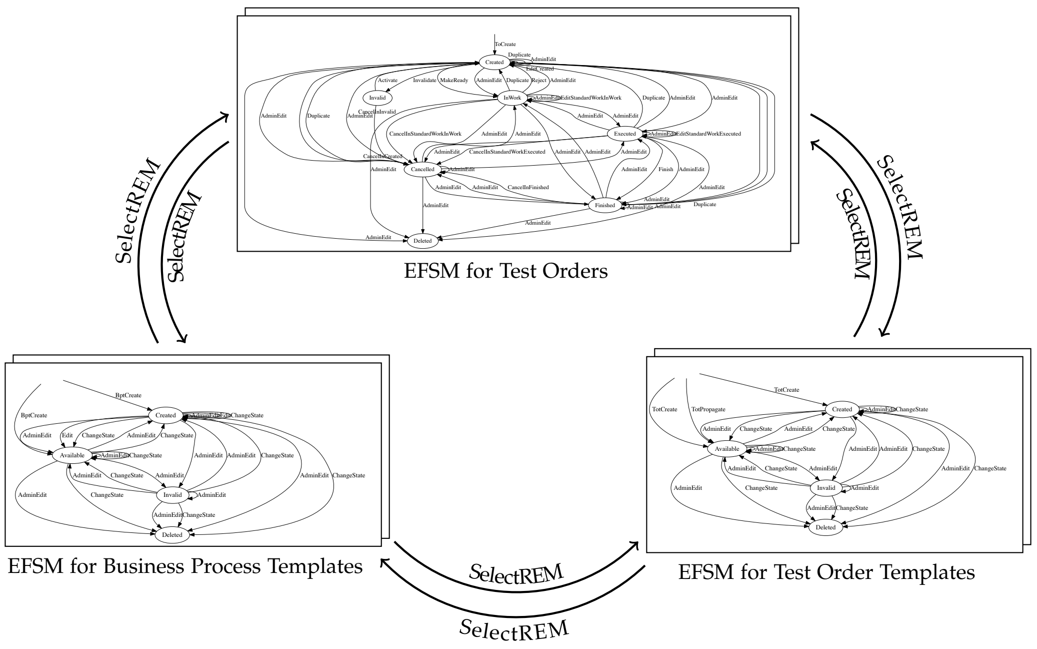


Figure 3.6: Switching between rule-engine models inside the Test Order Manager. For details about the EFSMs see Figures 3.9, 3.10 and 3.11.

3.5.1 Singleton Rule-Engine Models

As explained before, we parse the business-rule models from XML files and translate them into an EFSM. It should be noted that our model representation does not strictly follow the EFSM definition, because we used optimised data structures for the application of FsCheck. However, the semantics of our model combined with FsCheck correspond to an EFSM. While the translation to an EFSM (in Figure 3.4) provides the abstract syntax and the formal semantics of our models, we now focus on the concrete model implementation in the object-oriented context of FsCheck.

Our model is encoded as an object tree, i.e., an abstract syntax tree that serves as input to FsCheck as part of the Spec shown in Definition 3. The class diagram for this object structure is shown in Figure 3.8. It can be seen that the model consists of an attribute dictionary (i.e., a map), an initial state, a current state, a list of states and a dictionary of tasks. The transition relation is represented by a class called *Task* that contains hash sets for the possible source and target states of a task, a name and a flag, which indicates that the state should not change

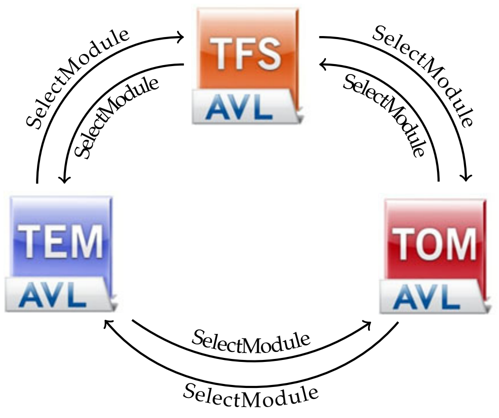


Figure 3.7: Switching between modules: Test Order Manager (TOM), Test Factory Scheduler (TFS), Test Equipment Manager (TEM).

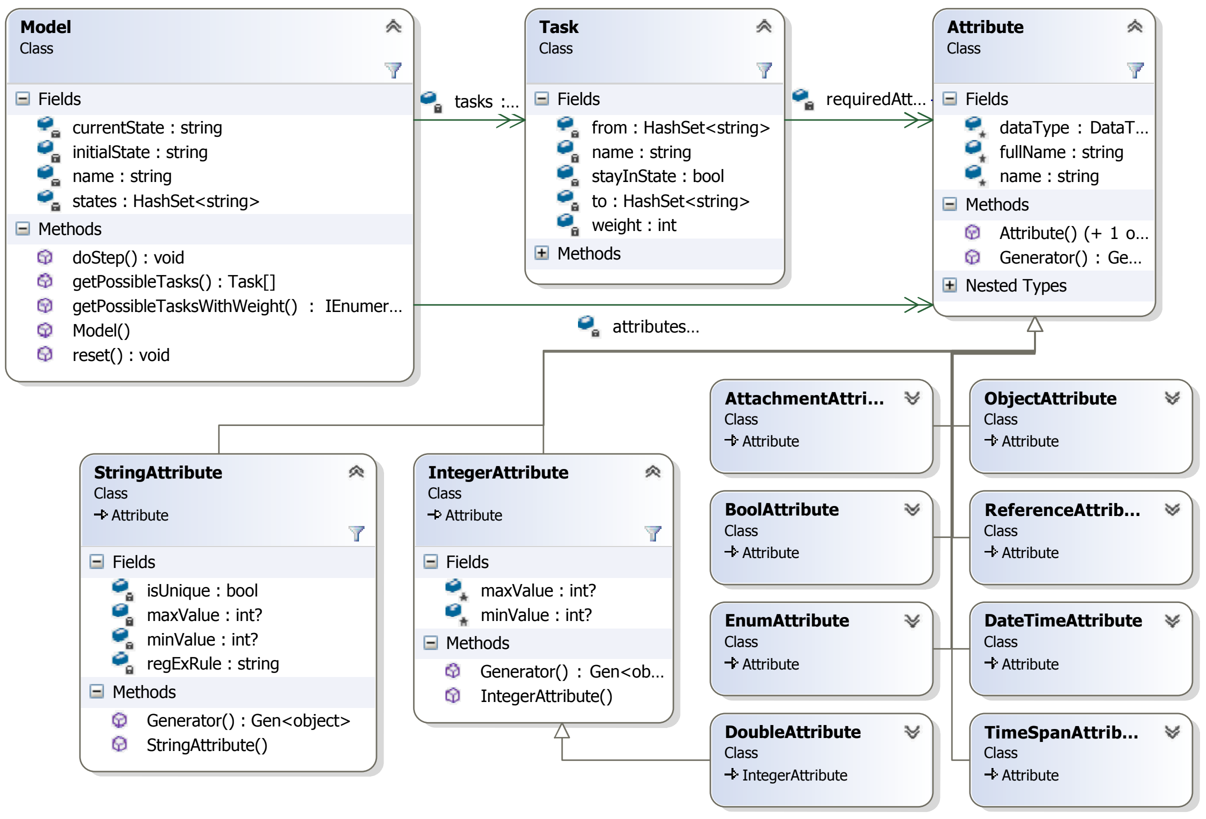


Figure 3.8: Class diagram for a model, which is parsed from XML and serves as input to FsCheck as part of the Spec.

after the execution of the task. Furthermore, the class includes a dictionary for the required attributes. The attributes represent the form data of a web-service operation. All attributes have a common base class, which has fields like name and data type. The derived classes for specific data types extend this base class by adding possible constraints and a custom generator for the data type that respects these constraints. For example, an integer attribute class can have constraints for the minimum and maximum value, and the generator chooses a number between these boundaries or an arbitrary number if no constraints are given. We have implemented attribute classes for simple data types, like enumerations, floating-point numbers, dates and times, but we also support more complex data types:

- Reference attributes: a reference to another object of the SUT can also be an attribute for a task. The possible options for this object are given by a query, which represents a search string for the database. The interface to the SUT provides a method to get results for a valid query and an element generator chooses one of the results randomly. This generator, was already explained in Section 3.3.2.
- Object attributes: an object attribute can group multiple attributes in a struct or a list. The generator for this type recursively calls the generators of included types, which can again be object attributes.
- Attachment attributes: some tasks require files of certain file types. The generator for this attribute chooses one of the possible file types and generates a random file name. The generation of the actual file is added to the wrapper class of the SUT, because the file should also be deleted after the test execution.
- String attributes: a string attribute may include restrictions like a minimum/maximum length or a regular expression. In order to generate strings that match these regular

Algorithm 5 Attribute data generation for the form data of the incident manager.

```
Input:
    Attributes: an array of attribute instances
Output:
    attributeData: a generator for maps (Attribute.Name $\to$ $Val$)
 1: function GenerateData(Attributes)
 2:     for each $attr \in Attributes$ do
 3:         $genArray.Add(attr.Generator())$        ▷ fill $genArray$ with Attribute Generators.
 4:     end for
 5:     return $Gen.Sequence(genArray).Select(Values \to ($
 6:         for $i \leftarrow 1$ to $length\_of(Attributes)$ do
 7:             $attributeData[Attributes[i].Name] \leftarrow Values[i]$
 8:         end for
 9:         return $attributeData))$
10: end function
```

expressions, we apply a .NET port of the Xeger library.[9] This library can generate text that matches a given regular expression.

The object representation of the model is also used for the interface specifications for FsCheck. For example, preconditions for the restriction of the tasks are automatically created by the model class. The generator for the next command also includes information of the model to generate commands with possible next states and attribute data.

Algorithm 5 shows how attribute data can be generated. First, an array of generators is created by iterating over the attributes and adding the generators to the array. This array is then given to a sequence generator as input, which creates an array of values for all the generators in the array (Line 5). In order to store them in a map, we use the select function of the generator. This function takes an anonymous function, which takes the values as input and returns an object that should be created by the generator. It can be applied to convert a generator of certain type A to a generator of a different type B by processing the generating values of the first generator. Hence, the select function has the following signature:

$$Gen[A].select : (A \to B) \to Gen[B]$$

In our case, we build a map generator from a sequence generator in order to enable the generation of maps with attribute names as keys and the data as values (Lines 6 to 9). This attribute data generation is required for the command generation, which is shown in Algorithm 6. First, an array of possible tasks is created in the model class, which considers the preconditions for this creation (Line 2). An element generator is used to choose one of these tasks and with the selectMany function we process the chosen task (Line 3). The selectMany function is similar to the select function. It can be applied to a generator and requires an anonymous function as argument. This anonymous function takes a value of the generator as input and has to return a new generator.

$$Gen[A].selectMany : (A \to Gen[B]) \to Gen[B]$$

Therefore, selectMany makes it possible to nest generators and also to pass the generated value to the inside generator.

A chosen task can lead to multiple next states, hence, we also choose a next state with an element generator (Line 4). Then, the attribute data generation of Algorithm 5 is applied. With the generated data we create a DynamicCommand object which takes the task, the model, the attribute data and the next state as arguments for the constructor (Line 6).

[9] `https://code.google.com/archive/p/xeger` (visited on 2018-09-19)

Algorithm 6 Next: generates a *Cmd* for a given model.

```
Input:
    model: model instance that incorporates the state
Output:
    gen: a generator for commands
 1: function spec.next(model)
 2:     $ts \leftarrow model.getPossibleTasks()$                          ▷ possible tasks
 3:     return $Gen.Elements(ts).selectMany(t \rightarrow$
 4:         $Gen.Elements(t.PossibleNextStates()).selectMany(s \rightarrow$
 5:             $GenerateData(t.requiredAttributes).select(data \rightarrow$
 6:                 new $DynamicCmd(t, data, s)))$
 7: end function
```

The outline of the DynamicCommand class is shown in Algorithm 7. This generic command class can handle the execution of all tasks of the parsed model. The class has the task, the model, attribute data and the next state as constructor arguments and as fields. They are required for the execution of the transition. Running a transition on the model is realised with a function of the model class (Line 5). The execution on the SUT in *runActual* calls the wrapper class of the SUT (Line 9). This wrapper class uses reflection to call the actual methods on the SUT and it also sets the attributes. In the postcondition, we check if the post-state of the model is equal to the post-state of the SUT. Note that our SUT allows an explicit observation of the state. This information can be accessed via the wrapper class of the SUT, which allowed us to easily compare the state of the model and the SUT in the postcondition. In other SUTs, this information might not be accessible, e.g., only the output might be comparable.

3.6 Evaluation

Our approach was developed for a web-service application called TFMS as discussed in Section 1.4.2. This system is a composition of various modules. We performed an evaluation of two representative modules of the application, the Test Order Manager and the Test Equipment Manager. Moreover, we also tested other small modules like the Incident Manager, which was shown in the example of Section 3.3.2, but the major part was the Test Order Manager. The goal of the evaluation was to analyse the applicability and bug-finding ability of our

Algorithm 7 DynamicCmd: generic *Cmd* definition.

```
Input:
    t : task instance,
    data : map (Attribute.Name → generated value),
    s : nextState (the selected next state from the next function)
 1: function post(sut, model)
 2:     return $sut.State = model.State$
 3: end function

 4: function runModel(model)
 5:     $model.doStep(t, s);$
 6:     return model
 7: end function

 8: function runActual(sut)
 9:     $sut.DefaultTask(t, data, s);$
10:     return sut
11: end function
```

Table 3.1: Number of states, tasks, transitions and attributes of the REMs within the Test Order Manager.

Model	States	Tasks	Transitions	Attributes
Test Order	8	16	49	15
Business Process Template	5	4	25	58
Test Order Templates	5	4	24	53
Test Order Manager	18	24	98	126

method for industrial use cases. The human effort of the evaluation was primarily the implementation of the parsing process of the business-rule models and the connection to the SUT. The human effort for the testing process itself was insignificant, due to the high automation.

We found several issues in the systems under test that are listed in the following sections.

3.6.1 Settings

We performed our experiments in a virtual machine with Windows Server 2008, 4 GB RAM and one CPU on a MacBook Pro (late 2013 version) with 8GB RAM and a 2.6 GHz Intel Core i5. The system was running TFMS 1.7 and we applied FsCheck 2.4 as PBT tool.

First, we ran the default test settings of FsCheck, which produces 100 test cases with an average length of 51. When an issue in the SUT had been found, we repeated the whole test process until no more issues were detected.

Additionally, we performed test runs with an increasing number of test cases and a fixed length of ten in order to evaluate the coverage on the models. This ensured that all relevant parts of the SUT were covered. Furthermore, this coverage analysis helped in regression testing. When fixing a detected issue, it gave us confidence that the randomly re-generated test sequences were covering the affected parts of the SUT. The results of this evaluation are presented in the following sections.

3.6.2 Test Order Manager Case Study

The Test Order Manager module controls individual work steps and preparations for automotive test orders and corresponding templates. A test order is a composition of multiple processes that are necessary for a test sequence at an automotive test field. It consists of both organisational and technical processes, which are defined in templates. Organisational processes, like accepting a test order, are defined in a Business Process Template, and a Test Order Template defines technical processes, like preparing a test cell on the test bed. For a test order we need to select both, a Business Process Template and a Test Order Template. Test Orders, Business Process Templates and Test Order Templates can be managed individually, and they have separate REMs as shown in Figure 3.6.

A state machine of a test order is shown in Figure 3.9. The figure displays only states and transitions, because there are too many attributes to show them. It can be seen that the model contains a number of states for the workflow or life cycle of a test order. The REMs of Business Process Templates and Test Order Templates are similar, but they have fewer states and transitions. They are illustrated in Figure 3.10 and Figure 3.11.

Table 3.1 displays the size of the models within the Test Order Manager module. It shows the number of states, tasks, transitions and attributes. It can be seen that the number of possible transitions is high. Therefore, our automated approach makes sense, because otherwise the test of all these transitions would be impractical, especially, since the transitions are not simple actions in this case study. Each transition represents the opening of a page, entering data for form fields and saving the page. One example page of an AdminEdit task can be

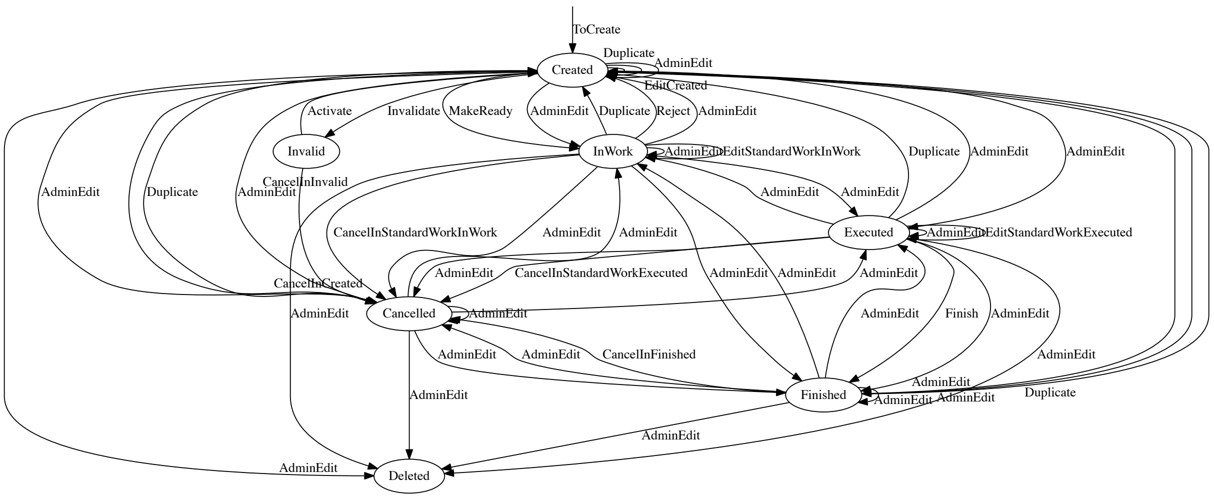


Figure 3.9: EFSM for Test Orders.

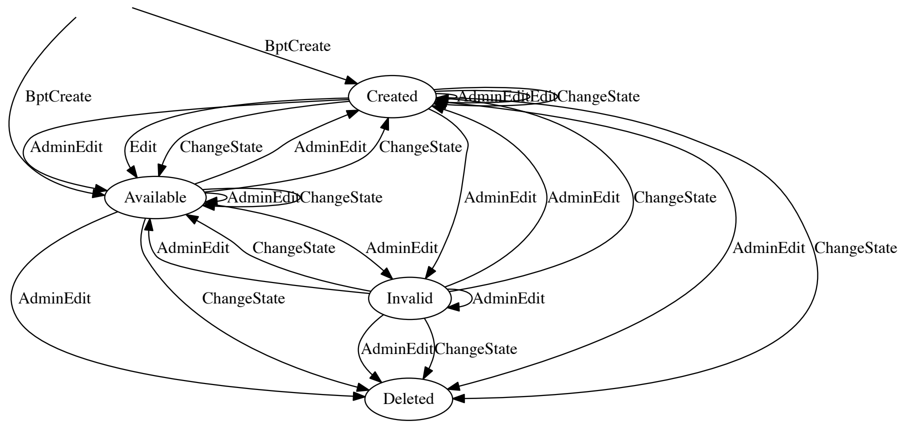


Figure 3.10: EFSM for Business Process Templates.

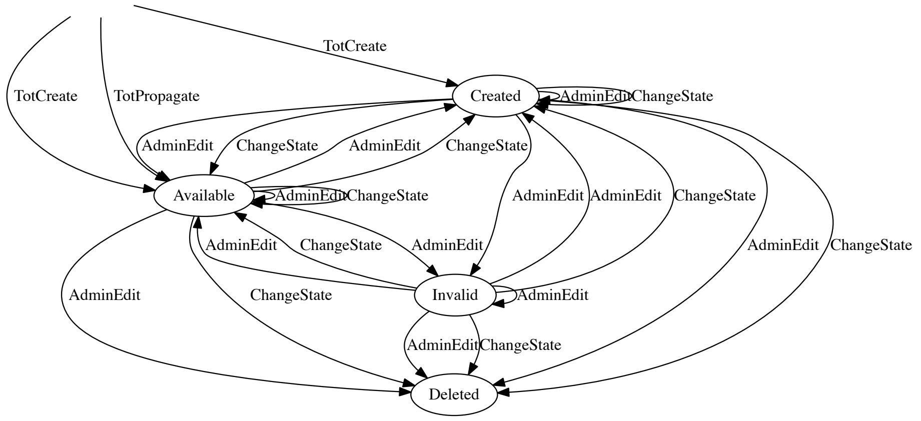
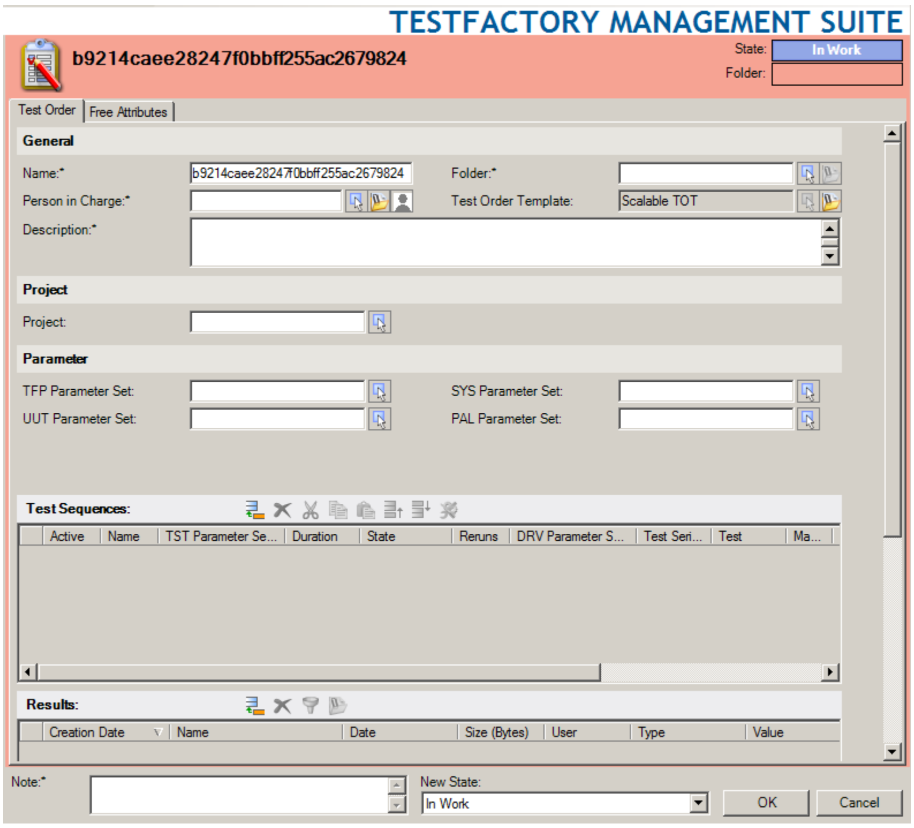


Figure 3.11: EFSM for Test Order Templates.

seen in Figure 3.12. This page is part of the graphical user interface of a client application that connects to web services on a server. It contains a number of form fields and tables that require generated data.

For the case study AVL provided us with a test framework that was specifically developed for the SUT and performs the communication with the web services on the server. It basically represents an abstraction of the graphical user interface and is intended to alleviate the testing effort. A tester should not need to know any web-service details in order to


TESTFACTORY MANAGEMENT SUITE

b9214caee28247f0bbff255ac2679824

State: In Work
Folder:

Test Order | Free Attributes

General

Name:* b9214caee28247f0bbff255ac2679824 Folder:*
Person in Charge:* Test Order Template: Scalable TOT
Description:*

Project

Project:

Parameter

TFP Parameter Set: SYS Parameter Set:
UUT Parameter Set: PAL Parameter Set:

Test Sequences:

Active	Name	TST Parameter Se...	Duration	State	Reruns	DRV Parameter S...	Test Seri...	Test	Ma...

Results:

Creation Date	Name	Date	Size (Bytes)	User	Type	Value

Note:* New State: In Work

Figure 3.12: TFMS form for the AdminEdit task.

run tests. Hence, the framework offers functions, which perform web-service requests in the background in order to execute the required steps of the test cases. The framework is written in C# and has interfaces to modules that provide functions for login/logout, executing tasks, opening domain objects, retrieving data and so on. We call these functions, e.g., to start tasks (representing the opening of forms), to set attributes (of form data), and to save forms.

The case study revealed the following problems and bugs:

1. There was a bug in the original testing framework that was provided by our industrial partner. The expected state after a task execution was sometimes wrong, because in some cases an old version of the object was used by the testing framework.
2. Another issue we detected concerns our test-case generation method. In some rare cases, the business models do not contain enough information. For example, there were reference attributes that could not be changed to a different subtype after an object was created. The query for these attributes needed an additional restriction for the subtype. This information was correctly implemented in the code, but is missing in the rule-engine model. Hence, the tool reported a bug that in fact was not a bug. It is rather a limitation of the approach of relying on the business-rule models as primary source for the test-case generation.

The following bugs were found in hidden tasks that were not enabled in the user interface. These tasks remained in the business-rule models, and they would cause problems when they were enabled again. Therefore, we also tested them.

3. There were tasks that first resulted in an exception, which stated that certain attributes are missing. However, when the attributes were set, it resulted in an exception that said that the attributes are not enabled.

Table 3.2: Average number of commands needed for finding the issues of the Test Order Manager.

Issue	Number of *Cmds*
1	-
2	1.4
3	23.8
4	9.4

4. There was a problem with a task that had a next state in the model, which was not permitted by the SUT. Furthermore, the error message of the SUT was wrong in this case. It should list possible next states. However, the list did not contain states, but tasks.

Table 3.2 presents the average number of commands that were needed to find the issues. The average was computed over five test runs. Note that for the first issue no data is available, as the bug in the testing framework was fixed in an early state of the evaluation. The data shows that Issue 3 is especially hard to find as it requires on average of more than 23 input commands to be executed until its detection.

We monitored the coverage of our tests on the model in order to obtain confidence that we tested thoroughly enough. The states and tasks of the model were covered with a few tests and are, therefore, omitted. The transition coverage for an increasing number of test cases is illustrated in Figure 3.13. The test case length is fixed to ten and the coverage is given as the average percentage of the transitions that are visited during 100 test runs with the same number of samples. Due to the high number of transitions we need about 4.000 test cases in order to obtain an average transition coverage that is close to 100%. We performed the same evaluation for transition-pair coverage, which is also called 1-switch coverage after Chow [46]. For this coverage criterion, we evaluate how many sequences of two consecutive transitions are observable within our test cases. The results are shown in Figure 3.14. Transition-pair coverage requires even more test cases, e.g., 5.000 test cases only produce an average transition-pair coverage of 75%.

3.6.3 Test Equipment Manager Case Study

Similar to the Test Order Manager we performed a case study for the Test Equipment Manager module. The main function of this module is the administration of all equipment that is relevant for the test field, like test beds, measurement devices, sensors, actuators and various input/output modules. All these test equipment can be created, edited, calibrated and maintained with the Test Equipment Manager. A hierarchy of test equipment types is used to classify the test equipment. Test configurations, which are compositions of different test equipment, can also be administrated and also the connection of devices via channels can be controlled with this module.

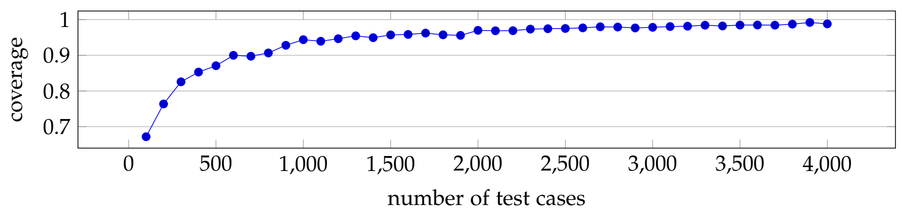


Figure 3.13: Test Order Manager: Transition coverage for increasing number of test cases (with test cases of length ten).

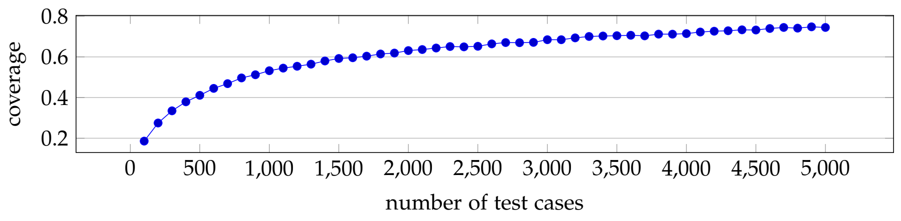


Figure 3.14: Test Order Manager: Transition-pair coverage for increasing number of test cases (with test cases of length ten).

Table 3.3: Number of states, tasks, transitions and attributes of the REMs of the Test Equipment Manager.

Model	States	Tasks	Transitions	Attributes
Test Equipment Type	5	10	21	43
Test Equipment	7	13	39	23
Test Equipment Manager	12	23	60	66

Figure 3.15 illustrates the main REMs and the complexity of the Test Equipment Manager. It can be seen that the EFSM for test equipment has many transitions for maintenance and administration purposes. Most of the state names are self-explanatory. The state *Invalid* describes an object that was copied and has to be adapted. *Mounted* is a state that means that the equipment was installed in the test field. The EFSM for test equipment types is smaller but similar, because it does not contain maintenance operations. Details about the behaviour of the REMs are omitted, because they are too specific for the SUT and not relevant for this work. The size of the module and its REMs is summarised in Table 3.3, which shows the number of states, tasks, transitions and attributes. In contrast to the Test Order Manager, we only have two REMs and the module is not as complex.

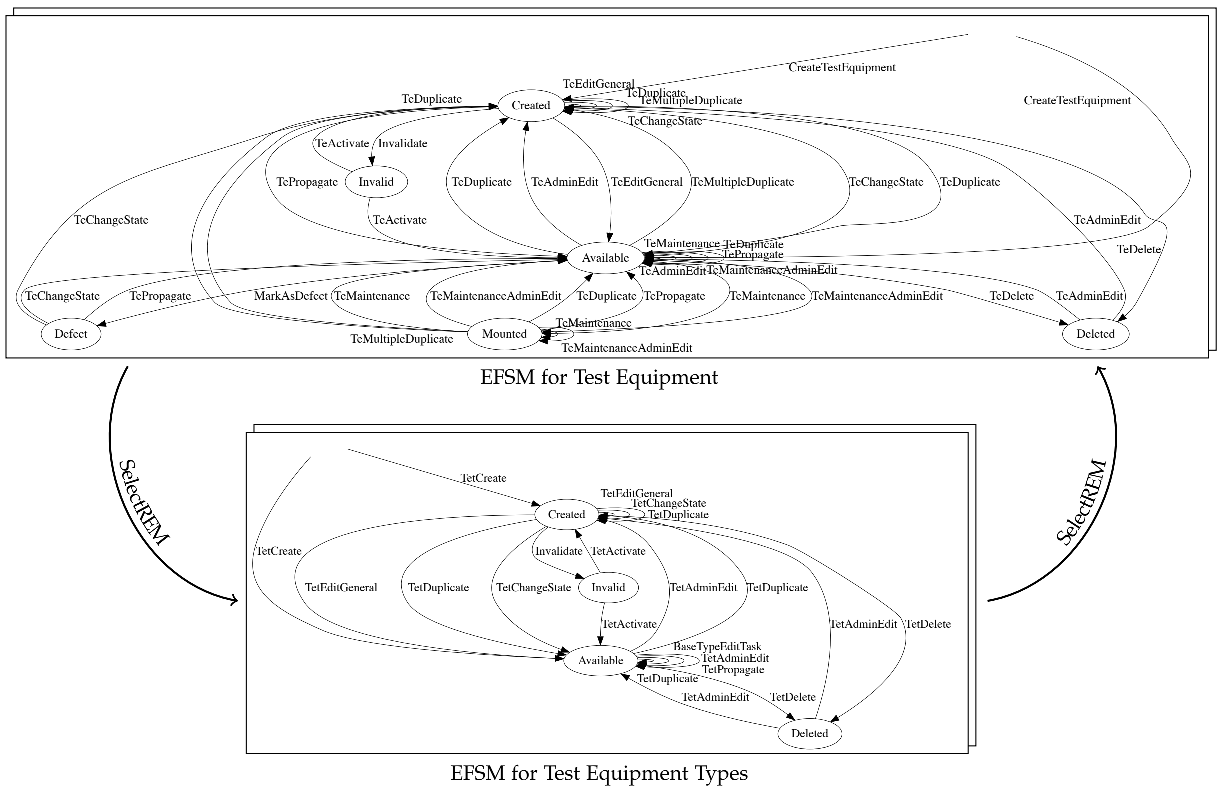


Figure 3.15: EFSM for the rule-engine models of the Test Equipment Manager module.

We found a number of issues which are listed below. It should be noted that the case study was performed with test rule-engine files, which are not used by actual customers and which were not inspected as intensively as productive rule-engine files. However, if productive rule-engines would contain these kinds of issues, then our tests could also find them. The following two issues could be found with strings by utilising our string generators, which support the generation of strings with regular expressions.

1. Inconsistency regarding the use of tab characters 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 database and only replaced, when they were sent to the graphical user interface. 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 found 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 at the beginning. However, the regular expressions were written so that they allowed all non-white space characters at the beginning and at 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 special characters were at the end of the string, then the name was not valid any more, 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.

Further 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, because 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 graphical user interface, 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 graphical user interface was possible.

Table 3.4 illustrates the average number of commands that were needed to find the issues. The numbers were computed in the same way as described for the Test Order Manager module in Section 3.6.2. The first issue was particularly hard to find. The reason is that the generator for strings does not generate a tab character very often, because it is only one of many options and the same string with a blank was also generated very rarely.

We also monitored the average transition and transition-pair coverage for an increasing test case number, as already shown for the Test Order Manager module. The results are shown in Figure 3.16 and Figure 3.17. Due to the lower complexity of this module, only about 150 test cases are needed to reach a transition coverage that is close to 100% and about 2.500 test cases for transition-pair coverage.

Table 3.4: Average number of commands needed for finding the issues of the Test Equipment Manager.

Issue	Number of *Cmds*
1	467.4
2	12.4
3	17.6
4	9

3.6.4 Further Result

Another bug was not directly found with our test cases, but during the extensions of our models with the select functionality as described in Section 3.4.2. In order to implement this functionality, we had to evaluate the behaviour of the SUT, because it is not described in the rule-engine models. During this evaluation we could observe a bug that occurred when we tried to open a window for a module while a task was executed in an already open window. In this case, the window crashed and it was not possible to open a new window for the module until it was terminated via the task manager. Note, this bug was not directly found with our automated method, but during the model design. However, in other MBT approaches, it is also often the case that bugs are found in this phase. Hence, finding such bugs can be seen as a positive by-product of applying MBT in general.

3.7 Property-Based Testing with External Test-Case Generators

We also showed how PBT can be extended in order to support other sources for the test-case generation instead of the default random walks on the model [10]. This gives the tester more control on how to produce meaningful command sequences for the test cases. For example, it can be applied to combine random testing with a mutation-based test-case generation method, which results in a powerful testing strategy [8].

By integrating an external test-case generator into a PBT tool, we can combine the dynamic test-data generation feature of PBT with the ability to generate command sequences (test cases) according to various coverage criteria.

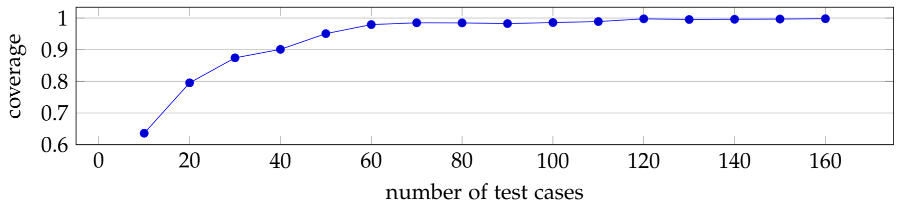


Figure 3.16: Test Equipment Manager: Transition coverage for increasing number of test cases (with test cases of length ten).

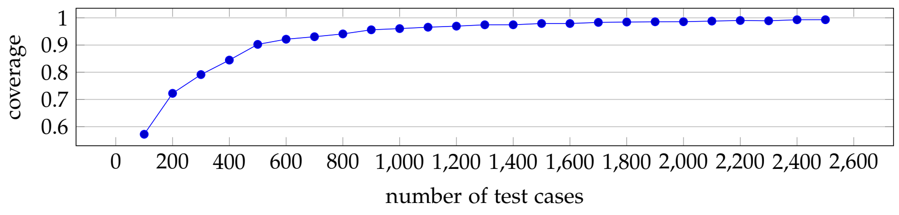


Figure 3.17: Test Equipment Manager: Transition-pair coverage for increasing number of test cases (with test cases of length ten).

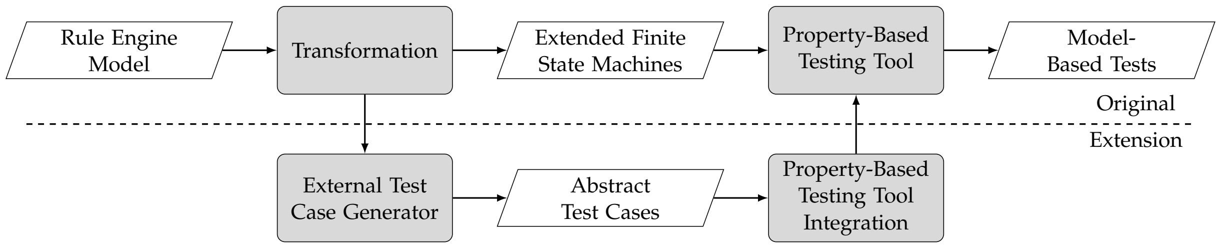

Rule Engine Model
Transformation
Extended Finite State Machines
Property-Based Testing Tool
Model-Based Tests
Original
Extension
External Test Case Generator
Abstract Test Cases
Property-Based Testing Tool Integration

Figure 3.18: Overview of the steps for the integration of an external test-case generator.

Figure 3.18 illustrates how we integrated an external test-case generator into our existing rule-engine-based testing technique. The first step works as already explained in the previous sections, i.e. we translate XML business-rule files to EFSM input models for FsCheck. As an alternative to directly using our EFSMs within FsCheck, we can also further transform the models in order to provide them to an external test-case generator. We applied the model-based mutation testing tool MoMuT::UML [102] in order to generate abstract test cases based on mutation coverage. This enabled the generation of smaller test suites that still cover important model parts.

An externally generated abstract test case can serve as input for a state-machine property, where it is executed instead of performing a random walk on the model. Additionally, we can enrich the command sequences from the external generator with test data generated with classical PBT. The reason why we do not also apply the external generators for the test-data generation is that they are often limited regarding the supported data types. Moreover, PBT is more suitable for this task, because it has powerful generators that can be combined very flexibly and thereby facilitate the generation of complex test data like attributes for web-forms. In order to execute externally generated abstract test cases, we extend the state-machine specification with new functionality for external data sources.

We evaluated this approach by applying MoMuT as an external test-case generator and by comparing it to FsCheck. We supplied MoMuT with observer automata that guaranteed a certain coverage of the model, like state or transition coverage. The most notable difference between the test generation with MoMuT and the random approach with FsCheck was the computation time. The test suite generation time with MoMuT is very high. It can take several minutes to hours to generate useful sequences, hence it becomes infeasible for bigger models.[10] This is a known limitation of the approach. FsCheck can generate a large number of test cases within one second.

We applied observer automata for MoMuT in order to perform a test-case generation that enables the fulfilment of a coverage criterion with just a single test case. Hence, the test suites generated with MoMuT were able to cover more of the model with fewer test cases as compared to plain FsCheck. This advantage becomes negligible as soon as the number and length of test cases is increased. However, 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 short MoMuT sequences cover most parts of the model and are therefore well suited for regression testing.

Implementations details and the results of the evaluation where presented in our previous work [10]. Additionally, a more detailed description of this method can also be found in the master's thesis of Silvio Marcovic [123].

[10]Note this only holds for the old version of this tool. MoMuT::UML has been reimplemented and the run time is not an issue any more [63, 131].

3.8 Discussion

3.8.1 Limitations and Threats to Validity

The evaluation demonstrated that our method of using business-rule models for PBT is able to find bugs in a real system. Moreover, we showed that our randomly generated test cases are able to achieve a high transition coverage with an acceptable number of tests. A limitation of random testing is that certain coverage criteria cannot be guaranteed and so important aspects of the SUT might not be tested. Hence, a more targeted test-case generation strategy might be able to find more bugs. We evaluated another strategy as explained in the previous section, but it was not able to find more bugs, because the random strategy already covered most of the model with just a few tests. Moreover, the random generation was especially helpful for the creation of complex form data, which was required for our SUT.

Another limitation of our approach is that we rely on business rules as test models and oracles. In an ideal implementation, we could only test if the business rules are interpreted correctly. However, for our SUT, we saw that there are a number of deviations of the SUT from the business rules. In other applications, this might not be the case. Furthermore, it should be noted that we are only able to find bugs that are caused from a deviation of the SUT from the business rules. A manually crafted model might be able to find more bugs, but it is expensive to create a model manually.

A limitation of relying on the business rules can also be that they might not contain enough information, e.g., not all data constraints that are present in the SUT might be encoded in the business-rule models. In such cases, a manual intervention might be needed. This was already mentioned in Issue 2 of the Test Order Manager case study in Section 3.6.2.

An external threat to the validity of our method is that the random generation of a PBT tool might not be random enough. For example, there can be problems when the random generation is based on the system time or when multiple threats share a common random generator. In order to eliminate this threat, we analysed our generated command sequences for suspicious patterns, like repeatedly occurring sequences. Moreover, we made sure that the random generation functionality was implemented according to common practice.

An internal threat to the validity of our evaluation might be the research bias, which can come in different shapes. (1) We might have selected an SUT that has particular faults in order to support our approach. However, we did not select the SUT for our evaluation. It was given to us by our industrial partner AVL and this was done before we had a particular testing method in mind. Hence, we had no influence over the choice of the SUT. (2) We could have found issues that are no real problems of the SUT. The fact that we had to present our findings to AVL and also that they had to confirm our found issues before we were allowed to publish them, dissolves this threat. (3) We could have targeted our testing method towards specific bugs that were present in the SUT. This would limit the type of faults that can be found, but we did not know the bugs of the SUT beforehand. They were revealed by our evaluation. Hence, it was not possible for us to target our testing method towards specific known bugs of the SUT.

Another internal threat to the validity is that we only tested our method with a specific system. One could argue that one case study is not enough to evaluate the applicability or generality of our method. However, we did evaluate two modules of one big web-service application. These modules have different functionality and can be used independently. Therefore, we think that the evaluation of two modules is sufficiently representative for this application domain.

3.8.2 Future Work

Additional case studies are an interesting option for future work. In order to analyse the generality of our method, it would make sense to test further applications that are driven by rule engines. Moreover, a comparison with other testing methods, like manual unit testing, would make sense. This might help to assess the bug-finding performance of our approach.

Another potential topic for future work would be fuzzing. With our current method, we only test the behaviour of the SUT that is allowed by the business rules. However, it would also be important to test behaviour that is outside the scope of the business rules, i.e., invalid behaviour. This could be done by specifying generators that generate data that is not allowed by the business rules, e.g., tasks that are not enabled in a specific state, or form data that does not meet certain restrictions. By generating invalid data, we could check if the error handling works as expected and also if the business rules are applied correctly.

3.9 Concluding Remarks

We have developed an automatic test-case generation approach for business-rule models of a web-service application. The approach is based on property-based testing and written in C# with the tool FsCheck.

First, we presented our business-rule models, and we introduced property-based testing, formalised its underlying concepts and algorithms. Next, we discussed how our approach works in detail. It takes XML files with the business-rule models as input and converts them into an EFSM in the form of an object representation that is used for FsCheck specifications and as model. We evaluated our approach in a case study with an industrial web-service application. Finally, we introduced an alternative test-case generation strategy that works with external test-case generators.

With our method, we were able to find eight issues that were confirmed by our industrial partner AVL. This demonstrates the effectiveness of our approach.

In the next chapters, we will see how this method can be applied for load testing.

4 Integrating Statistical Model Checking Into Property-Based Testing

This chapter is primarily based on our publications at MEMOCODE 2016 [7] and ICST 2017 [6]. Some small parts are from our papers at QEST 2018 [3] and in the SQJO journal 2017 [9].

4.1 Overview

In recent years, SMC (Section 2.2) has become increasingly popular, because it scales well to larger stochastic models and is relatively simple to implement. In this chapter, we show how SMC can be easily integrated into a PBT framework, like FsCheck for C#. As a result we obtain a very flexible testing and simulation environment, where a programmer can define models and properties in a familiar programming language. The advantages are that no external modelling language is needed and that both, stochastic models and implementations, can be checked. In addition, we have access to the powerful test-data generators of a PBT tool. We demonstrate the feasibility of our approach by repeating three experiments from the SMC literature.

A number of tools exist that perform SMC for different kinds of models. For example, UPPAAL-SMC checks priced timed automata [42] or PLASMA-lab supports a number of different modelling languages, like the Reactive Module Language or Matlab Simulink [38, 94]. However, the existing SMC tools are not as flexible as some situations or users may require. They are limited by the modelling language and the properties are limited by the used logics. Therefore, we propose a new SMC approach that builds on PBT.

For our SMC approach, we introduce new SMC properties that are integrated into a PBT tool. These SMC properties take a PBT property, configurations for the PBT execution, and parameters for the specific SMC algorithm as input. Then, our properties perform an SMC algorithm by utilising the PBT tool as simulation environment, and they return either a quantitative or qualitative result, depending on the algorithm. Figure 4.1 illustrates how we evaluate a PBT state-machine property within an SMC property.

With our approach we can do both, SMC by simulating stochastic models and conformance testing of an SUT with stochastic failures. For classical SMC, we evaluate our stochastic models with PBT state-machine properties, but we only exploit the model part of these state-machine properties, the part for the SUT is neglected. Additionally, conformance testing can be done by utilising the default state-machine properties for comparing faulty systems with a correct model within our SMC properties. This allows us to analyse stochastic failures of an SUT, e.g., we can estimate the probability of the occurrence of a failure with a Monte Carlo simulation.

PBT provides the tester with generators that enable the generation of test data with certain probability distributions. For example, it is possible to choose between multiple transitions by assigning weights to each of them. The default behaviour for checking PBT state machines

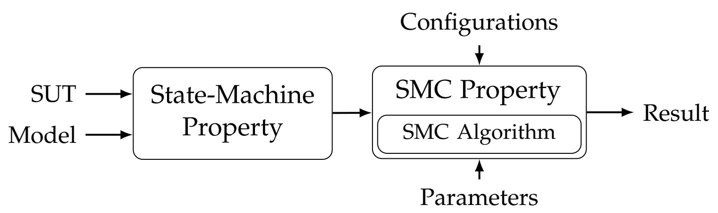


Figure 4.1: Data flow diagram of an SMC property.

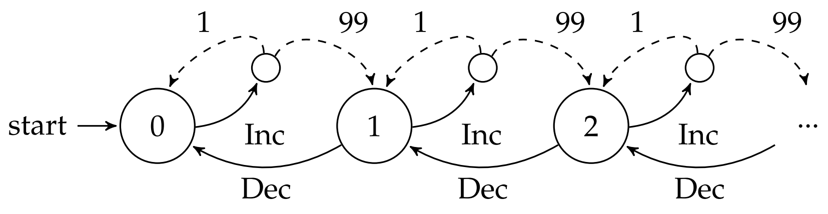


Figure 4.2: Stochastic model example of a counter.

is to make random walks through the model by generating (input) command sequences. The generation of these sequences can also be controlled with custom generators. For SMC, we need a discrete-event simulation, which can be realised via the random walks in PBT. Hence, PBT has a number of features that are helpful to implement a statistical model checker. For the demonstration of our approach we use the PBT tool FsCheck and C# as a programming language.

The contributions of this chapter are the following:

1. The main contribution is a new SMC approach that uses the modelling notations from PBT and checks PBT properties instead of logical formulas that are used in conventional SMC approaches. It can also be seen as a novel extension of PBT with SMC functionality providing testers who are already familiar with a PBT tool the option to analyse the stochastic properties of their SUT.
2. We present the application of our approach for an assessment of stochastic failures of an SUT by a comparison with an ideal model.
3. Moreover, we present an optimised PBT approach for classical SMC. The optimisation is that we only exploit the model part of a state-machine property in order to avoid the overhead of running both a model and an SUT, and it also gives us the possibility to stop during the generation of a sample.
4. We evaluate this approach by repeating three typical SMC examples from the literature.

The rest of this chapter is structured as follows. First, in Section 4.2 we demonstrate how SMC methods can be applied to a small example of a stochastic counter with faulty behaviour. Then, in Section 4.3 we present the implementation details. In Section 4.4, we evaluate our approach. Finally, we draw our conclusions in Section 4.5.

4.2 Example

In this section, we demonstrate our approach with a simple example of a counter, which is commonly used in the PBT community in order to illustrate model-based testing with state machine properties.

Figure 4.2 shows the state machine of our counter implementation. It can be seen that we added stochastic faulty behaviour to the increment function (*Inc*) of the counter. This behaviour was achieved by adding a probabilistic choice: the function can either do a normal increment (99%) or do nothing (1%). The decrement function (*Dec*) works as usual.

Algorithm 8 shows the implementation of the counter with the stochastic behaviour. A *Random* instance *rand*, which is a pseudo-random number generator from the .NET framework, is given as input and allows us to implement the stochastic behaviour. The counter has an integer *s* to store the state, which is initialised to zero (Line 1). In the *Inc* function, we call *random.Next*(100), which gives us numbers from 0 to 99. If the number is greater than zero, then we perform a normal increase, otherwise no increase occurs which represents a failure. The *Dec* function works as usual and there is a *Get* function to obtain the value of the counter.

Algorithm 8 Stochastic counter implementation for FsCheck.

```
Input:
    rand: Random instance (from .NET) for the generation of random numbers,
 1: $s \leftarrow 0$                         ▷ $s \in \mathbb{N}$ is a number for the internal state of the counter
 2: function Inc
 3:     if $rand.Next(100) > 0$ then          ▷ generate a random number between 0 and 99 with a uniform distribution
 4:         $s \leftarrow s + 1$
 5:     end if
 6: end function
 7: function Dec
 8:     if $s > 0$ then
 9:         $s \leftarrow s - 1$
10:     end if
11: end function
12: function Get
13:     return $s$                            ▷ the internal state is returned for an external inspection
14: end function
```

The main property we wanted to check for this example is how likely it is that the counter with the stochastic behaviour behaves like a normal counter. In order to check such properties, we implemented new properties that are based on the properties from PBT with the difference that they perform an SMC algorithm instead of normal property checks. Our new SMC properties take a normal PBT property and parameters for an SMC algorithm as input and apply this SMC algorithm on the input property. More details about the implementation of these properties are discussed in Section 4.3.

The state-machine specification for such a counter is similar to the one illustrated in Listing 2.1 in Section 2.1.3, with the only difference that we have to set a fixed length for each sample (or test case). This length can be supplied through the constructor of the state-machine specification.

We can evaluate our counter with a property for a Monte Carlo simulation as explained in Section 2.2.1 as follows:

```
Property p = new CounterMachine(10).ToProperty();
new MonteCarloProperty(p, config, 1000).QuickCheck();
```

It can be seen that we first define an FsCheck state-machine property, and we set the sample length to ten commands with the constructor argument. Then, we check this property by performing a standard Monte Carlo simulation with 1000 runs. This is done by defining a MonteCarloProperty that takes the state-machine property and configuration parameters as input and executing the *QuickCheck* method. The output of this method was that the property holds in 94.5% of the cases.

Another example of a property for the sequential probability ratio test (SPRT) that was introduced in Section 2.2.3 is shown in the following listing:

```
new SPRTProperty(p, config, 0.95, 0.9, 0.01, 0.01)
```

The four arguments of the SPRT method are: the probability for the null hypothesis, the probability for the alternative hypothesis and the type I and type II error parameters. The example demonstrates an SPRTProperty, which can check if the probability that the stochastic counter works like a normal counter is closer to 0.95 or 0.9. When we check this property for samples of length 10, we obtain the result that the null hypothesis H_0 was accepted, which means that the probability was closer to 0.95.

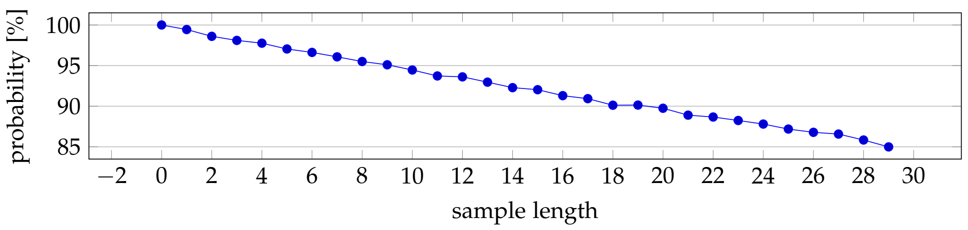


Figure 4.3: Simulation results for the property: how likely is it that the stochastic counter behaves like a normal counter?

Similar to the SPRT, a property for the Cumulative Sum (CUSUM) as discussed in Section 2.2.4 can be defined as follows:

```
new CusumProperty(p, config, 0.945, 0.85, 5, 5000)
```

This property also requires four arguments: the initial probability, the probability to detect a change, the sensitivity threshold and a maximum sample number to stop, when no change was detected. When we run this CusumProperty with samples of length 10, then, as expected, no change is detected. After the model was adapted so that the probability of a correct increment is decreased once after 1000 *Inc* commands, we were able to observe this change after 345 samples. CUSUM is useful for testing systems with random failures where the probability of failure changes after a while. Knowing when the change occurs helps at localising the fault.

Figure 4.3 shows the evaluation results of our counter that were obtained in the same way as explained for the MonteCarloProperty example, but with various sample lengths. We performed a Monte Carlo simulation with 100,000 samples for each data point in order to compute the probability that the stochastic counter acts like a normal counter. It can be seen that the probability decreases with increasing sample length. This behaviour met our expectations because with a larger sample length, i.e., longer random walks on the model, it is more likely that we see a faulty increment.

The concrete testing of properties that we want to check happens in the state-machine specification of FsCheck. We propose the following optimised approach for SMC. In conventional PBT, a state-machine property has a part for the model and for the SUT, which are both executed and compared. Due to the overhead of running both the model and the SUT, we utilise only the model part of these properties to simulate stochastic models, the SUT part is ignored. We instrument the model with observer functions that monitor the state of the model during execution. With these observer functions we form the conditions that are checked during run time (run-time verification). These conditions can be directly inserted in the *runModel* function of a command, which is responsible to perform the execution of an action on the model. By performing checks in the *runModel*, we can stop the sample execution (or test-case generation) prematurely, when we observe that the property fails. If we see that the property is already fulfilled, we can also terminate the sample execution with a stop command.

Additionally, it is also possible to make a conformance test between an ideal model and an SUT with stochastic failures. In this setting, we provide the default state-machine property of FsCheck as input to our SMC properties. As mentioned, the default state-machine property runs both a model and an SUT and checks, if the state of the SUT conforms to the model. For example, we can use the stochastic counter as SUT and a regular counter as model and test if we can find a difference for a certain number of generated command sequences. This approach is useful if an SUT has failures that occur irregularly or if a black-box system is tested that cannot be easily instrumented with additional observer functions. In the following, we will apply our optimised SMC approach, i.e., only using the model part, because it is better suited for classical SMC.

Algorithm 9 Pseudo code of the MonteCarloProperty.

```
Input: prop: PBT property, config: configuration for the property check, n: number of samples
1: function QuickCheck
2:     $passCnt \leftarrow 0$                          ▷ Counter for the passed property checks
3:     for $i \leftarrow 1$ to $n$ do
4:         if $prop.Check(config)$ then
5:             $passCnt \leftarrow passCnt + 1$        ▷ increase pass counter
6:         end if
7:     end for
8:     return $passCnt/n$
9: end function
```

4.3 Implementation

In this section, we illustrate how we implemented our SMC approach by introducing SMC properties that are based on PBT properties. Furthermore, we want to highlight the advantages like flexibility and user convenience of our proposed approach. The mathematical background of our implemented SMC algorithms was already briefly discussed in Section 2.2.

We propose properties for each SMC algorithm. These properties are based on properties from PBT with the difference that they perform an SMC algorithm instead of a normal test that only checks if a property holds or fails. We want to know the probability that a property holds, and we want to assess which of two given probabilities is closer to the probability of the property. Our SMC properties take a normal PBT property, a configuration object for the check of the PBT property and parameters for an SMC algorithm as constructor arguments. They provide a *QuickCheck* function that performs the SMC algorithm by simulating the input PBT property, which is used to generate samples and also to evaluate them. When the simulation is finished, the result is presented to the user. The SMC properties for the different SMC algorithms have the same structure, but they require different parameters for the algorithms and different stopping criteria for the simulation.

A simple code example of an SMC property that performs a Monte Carlo simulation is outlined in Algorithm 9. This SMC property takes a PBT property, configurations for the property check and the required number of samples n as input. First, we initialise a counter for the number of passing samples *passCnt*. Then, we run a for-loop that creates samples with the specified number of samples. The actual evaluation is done with the *Check* method of the PBT property, which takes the *config* object as input and generates a sample. A *config* object contains FsCheck configurations like Boolean flags to control the output/exception behaviour of properties and the required number of tests. The result of the *Check* method is true, if the property was fulfilled and false otherwise. In the case that it was true, we increase the counter for the passed samples. Finally, after the desired number of samples was evaluated, the result is the value of this counter divided by the total number of samples.

Algorithm 10 shows the pseudo code of a *ChernoffProperty*, which performs a Monte Carlo simulation with Chernoff-Hoeffding bound as described in Section 2.2.2. It can be seen that this property is very similar to a MonteCarloProperty of Algorithm 9. The only difference is that for the standard Monte Carlo simulation we need the total number of samples as input. For the version with Chernoff-Hoeffding bound, we need to specify the required accuracy and confidence with the parameters epsilon and delta. Then, the algorithm computes the required number of samples and performs a Monte Carlo simulation. Due to the fact that this property is so similar to a MonteCarloProperty, we can reuse the behaviour by inheritance. In our object-oriented setting, we can derive from the MonteCarloProperty and only add the additional calculation of the required number of samples (Line 2). All other steps are the same.

Algorithm 10 Pseudo code of the ChernoffProperty.

```
Input: prop: PBT property, config: PBT configuration, $\epsilon$: required error bound, $\delta$: confidence parameter
 1: function QuickCheck
 2:     $n \leftarrow \left\lceil \frac{1}{2\epsilon^2} \log \frac{2}{\delta} \right\rceil$        ▷ Calculate the required number of samples
 3:     $passCnt \leftarrow 0$        ▷ Counter for the passed property checks
 4:     for $i \leftarrow 1$ to $n$ do
 5:         if $prop.Check(config)$ then
 6:             $passCnt \leftarrow passCnt + 1$        ▷ increase pass counter
 7:         end if
 8:     end for
 9:     return $passCnt/n$
10: end function
```

Algorithm 11 shows an SPRTProperty that performs the sequential probability ratio test (Section 2.2.3). The inputs of this algorithm are a PBT property, configurations for PBT, probabilities for H_0/H_1 and the type I and type II error parameters α, β. The algorithm produces samples (Line 4) and calculates the log likelihood ratio (Line 5 & 7) repeatedly, until we are outside the indifference region, which is defined by α and β (Line 9). Finally, when we are outside the indifference region, we return H_1 as result, when the ratio is below the lower bound and H_0 otherwise.

A CUSUMProperty that performs the CUSUM algorithm (Section 2.2.4) is illustrated by Algorithm 12. As parameters this property requires a PBT property, configurations for PBT, an initial probability *p_init*, a probability k for detecting a change, a sensitivity threshold λ, and a maximum number of samples n for stopping when no change was detected. The first steps of the algorithm are the same as for the SPRT, because we also need to calculate the log likelihood ratio (Lines 6 & 8). The difference is that we calculate the minimum of the ratio sums (Lines 10–14) and check if the difference to the current value is greater than λ in order to detect a change (Lines 15–17). This inspection is done inside a loop over the maximum number of samples n. If no change was detected and the loop is finished, then the algorithm produces a corresponding output (Line 19).

The architecture of our SMC properties makes it easy to check all kinds of PBT properties. Although our focus is on stochastic models and state-machine properties, it is also possible to check the stochastic behaviour of other kinds of properties, like algebraic properties. For

Algorithm 11 Pseudo code of the SPRTProperty.

```
Input: prop: PBT property for producing a sample, config: configuration for checking the property with
    PBT, $p_0, p_1$: probabilities for $H_0$ and $H_1$ $\alpha, \beta$: type I and type II error parameters
 1: function QuickCheck
 2:     $ratio \leftarrow 0$
 3:     do
 4:         if $prop.Check(config)$ then        ▷ produces sample and checks result of PBT property
 5:             $ratio \leftarrow ratio + log(\frac{p_1}{p_0})$        ▷ calculate the log likelihood ratio
 6:         else
 7:             $ratio \leftarrow ratio + log(\frac{1-p_1}{1-p_0})$        ▷ calculate the log-likelihood ratio
 8:         end if
 9:     while $\frac{\beta}{1-\alpha} < ratio \wedge ratio < \frac{1-\beta}{\alpha}$        ▷ stop when threshold was reached
10:     if $ratio \geq \frac{1-\beta}{\alpha}$ then
11:         return $H_1$        ▷ $H_1$ is accepted
12:     else
13:         return $H_0$        ▷ $H_0$ is accepted
14:     end if
15: end function
```

Algorithm 12 Pseudo code of the CUSUMProperty.

Input: *prop*: PBT property for sampling, *config*: configuration for checking the property with PBT, p_{init}: initial probability without a change, k: probability that represents a change λ: sensitivity threshold n: max. number of samples for the algorithm

```
1: function QuickCheck
2:     $S_i \leftarrow 0$
3:     $min \leftarrow 0$
4:     for $i \leftarrow 1$ to $n$ do
5:         if $prop.Check(config)$ then
6:             $S_i \leftarrow S_i + log(\frac{k}{p_{init}})$        ▷ calculate the log likelihood ratio
7:         else
8:             $S_i \leftarrow S_i + log(\frac{1-k}{1-p_{init}})$        ▷ calculate the log-likelihood ratio
9:         end if
10:        if $i = 0$ then
11:            $min \leftarrow S_i$
12:        else
13:            $min \leftarrow Min(S_i, min)$
14:        end if
15:        if $S_i - min \geq \lambda$ then
16:            return "Change detected after " ++ $i$ ++ " samples!"
17:        end if
18:    end for
19:    return "No change detected after " ++ $n$ ++ " samples!"
20: end function
```

example, one might want to check properties of a stochastic function or a call to an operation with stochastic failures. Our properties can easily be implemented in other PBT tools. As already explained in Section 2.1, there exist various PBT tools for different programming languages. It does not require much effort to apply our approach for other tools since the structure is simple and works for other languages as well.

The definition of stochastic models and properties in a high level programming language provides some benefits like flexibility. For example, the models can be easily extended to include observer functionality like counting certain incidents. Counters can then be evaluated within the FsCheck specification in order to decide if a sample fails. We looked at existing SMC approaches and noticed that they are quite limited in some areas, e.g., if one wants to check models with different numbers of instances or if instances should be created dynamically. In a high-level programming language, it is quite easy to create a fixed number of instances via a loop or even dynamically add instances during the execution of a model. Furthermore, we noticed that often very long formulas are required for the properties within the models of existing SMC approaches, because the used notations often do not support loop functionality. We will give examples and further details about these issues in Section 4.4. It should be noted that we used a new experimental version of the FsCheck state machine specification. The advantage of this version is that it enables the generation of samples with a fixed length and that it supports stop commands, which allows us to stop during the command generation. These two features are quite important for our implementation, because we have to ensure that the generated samples are long enough, but it is also important that we can stop, when we know the result of a sample. More details about this new experimental version can be found in the documentation.[11]

[11] `https://fscheck.github.io/FsCheck/StatefulTestingNew.html` (visited on 2018-09-19)

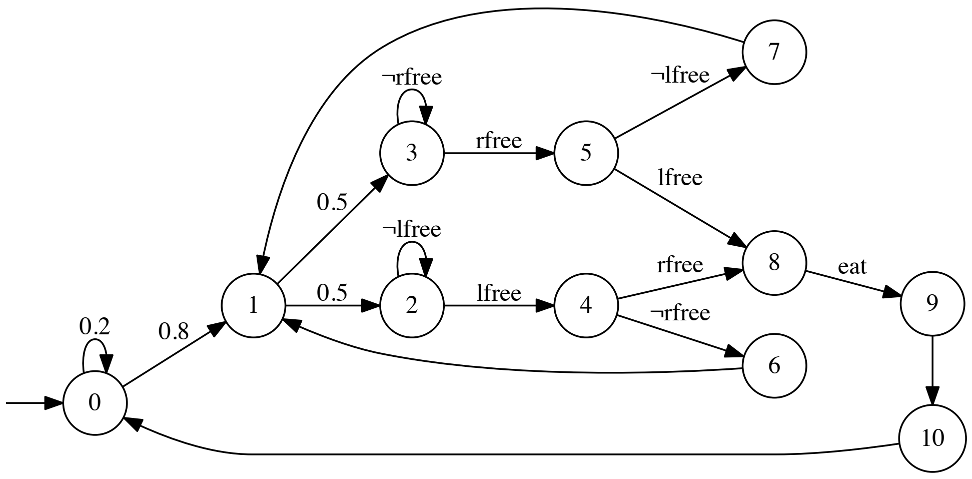


Figure 4.4: State machine of a philosopher as presented for PLASMA-lab.

4.4 Evaluation

In this section, we evaluate our SMC approach by applying it to three existing case studies from the SMC community and discuss differences to PLASMA-lab. We report performance results, because they formed part of an original PLASMA-lab case study. However, our primary focus is not performance, but the usability and flexibility of our modelling style.

4.4.1 Dining Philosophers Case Study

In a first case study, we applied our approach to a probabilistic version of the *dining philosophers* by Pnueli and Zuck [147]. We based our implementation on a case study which was presented on the PLASMA-lab website.[12] A similar example was also shown for PRISM [105].

The implementation for this example was straightforward. We have a simple philosopher state machine, which is illustrated in Figure 4.4. A philosopher first decides if he wants to remain thinking or if he becomes hungry. In the States 1 – 7 he is hungry and in States 8 and 9 he is eating. The guards *lfree* and *rfree* determine if the left and right fork are free. Our model is basically a circle of individual philosophers which all have a right and left neighbour, but it also contains observation and control functionality like a counter for steps and Boolean variables to check, if someone was eating in the past. A generator serves as a scheduler that randomly selects a philosopher that should be executed or generates a stop command when we know the outcome of a sample. The generator is part of the FsCheck state-machine specification that serves as our simulation environment. Only one command class is needed for this specification, which is responsible for the execution of the model and also performs the evaluation of our properties.

We checked the same quantitative properties as used for the PLASMA-lab case study.

1. What is the probability that any philosophers will be hungry within 1000 steps and that any philosopher will eat within 1000 steps after a philosopher was hungry?
2. What is the probability that a given philosopher will eat within 30 steps (for a table size of 150)?

We performed our evaluation in a virtual machine with 4 GB RAM and one CPU on a MacBook Pro (late 2013 version) with 8 GB RAM and a 2.6 GHz Intel Core i5. The first property

[12]`https://project.inria.fr/plasma-lab/examples/dining-philosophers` (visited on 2018-09-19)

Table 4.1: Dining philosophers run time comparison for rising table size for Property 1.

#Philosophers	Resulting Probability	Run Time [s]	PLASMA-lab Run Time [s]
3	1	969	6
10	1	991	14
30	1	1031	47
100	1	1145	256
300	1	1538	2151
1000	1	2676	17,057

was evaluated for different numbers of philosophers by applying a Monte Carlo simulation with Chernoff-Hoeffding bound. The parameter settings were as follows: $\epsilon = 0.003$ and $\delta = 0.01$, which results in a required number of samples of 294,351. The property was checked with our approach and with PLASMA-lab version 1.4.0 with the same parameters and thus the same number of samples.

The results are shown in Table 4.1. The property was always true both with our technique and with PLASMA-lab. For philosopher tables with a small size our approach is slower than PLASMA-lab, but for a larger number of philosophers our approach performs better. We assume the reason for this is that we can check the property in a more efficient way. We do not always check if all philosophers become hungry or are eating, we only check the currently executed philosopher.

We checked the second property with a Monte Carlo simulation with 30 million samples. The property was true for 29 samples, which means the probability is 9.6×10^{-7}. The run time was 110 minutes. The results are similar to those of PLASMA-lab. In contrast to them, we used a smaller number of samples, and we did not implement parallelisation.

Additionally, we checked two qualitative properties:

1. Is the probability that a given philosopher will eat within 50 steps closer to 0.1 or 0.15 (for a table size of 20)?
2. Can a change in the probability that a given philosopher eats within 50 steps be detected, when the number of philosophers rises? (We start with a certain number of philosophers and add a philosopher every 300 samples.)

We checked the first property with the SPRT with value 0.01 for the type I and type II error parameters (α and β). The result was that the alternative hypothesis H_1 was accepted, which means that the probability that a given philosopher will eat within 50 was closer to 0.15.

The second property was evaluated with the CUSUM algorithm with different initial numbers of philosophers. Initially, we performed a Monte Carlo simulation to obtain the probability p_{init} for a constant number of philosophers. Then, we adapted the original model so that a new philosopher was added every 300 samples. We wanted to detect when the probability is 10% below the initial probability, which gives us the threshold $k = p_{init} - 0.1$. For the sensitivity threshold we selected the value eight, which was enough to prevent false positives, and we chose 5000 as a maximum number of samples. The results are shown in Table 4.2. It can be seen that a change can be detected quite fast, for example, for five philosophers we can detect a change after 319 samples, when the number of philosophers was increased to six. For a higher initial philosopher number the CUSUM algorithm takes longer, because the probability change is smaller when one philosopher is added.

We compared our modelling style in the programming language to the models defined for PLASMA-lab and noticed several differences. PLASMA-lab needs separate models for settings with different numbers of philosophers. We have only one model that contains a parameter for the table size. Furthermore, our model supports a dynamic change of the number of philosophers. Another observation are the long formulas in the PLASMA-lab models. The models contain formulas that include variables for each philosopher, e.g.:

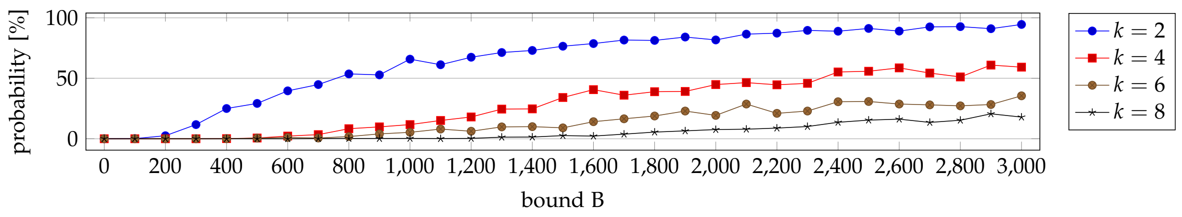


Figure 4.5: Simulation results for the property: can the protocol finish within B steps for different k values and a process number of 10?

```
label "hungry" = ((p1>0)&(p1<8))|...|((pn>0)&(pn<8));
```

For a model with 300 philosophers this quickly becomes impracticable. This can be avoided with an abstraction layer added to PLASMA-lab, e.g., by introducing a custom DSL [16]. In contrast, modelling in a programming language allows us to formulate these (quantified) formulas with loops. Consequently, in our object-oriented framework it is very easy to create models and to adjust them to different settings. For example, the philosopher case study was implemented within an hour and it can be easily adjusted to different settings.

On the other hand, PLASMA-lab provides a nice graphical user interface that helps the user to become familiar with the SMC techniques. Moreover, it provides a helpful simulation feature for debugging, which makes it possible to execute a model step by step and inspect all variables.

4.4.2 Randomised Consensus Case Study

The second case study is the *randomised consensus shared coin protocol* by Aspnes and Herlihy [23]. Our model is inspired by a PRISM case study [104]. It is also a PLASMA-lab case study, which is presented at its website.[13] The protocol describes an algorithm for achieving consensus among a number of processes that can communicate via shared memory. The protocol needs a constant parameter k that is required for the computation and influences the probability that the protocol finishes within a certain number of steps B.

The results of our case study are presented in Figure 4.5. Each data point in this figure was computed with a Monte Carlo simulation with 1000 samples. The fluctuations could be avoided with a larger number of samples. We performed simulations with PLASMA-lab in order to compare our results. The results of our SMC approach are consistent with the results obtained when running PLASMA-lab.

Additionally, we applied the SPRT in order to check the following property: is the probability that the protocol finishes within 500 steps closer to a null hypothesis 0.2 or to an alternative hypothesis 0.3, when we consider $k = 2$ and ten processes? We used the value 0.01 for the type I and type II error parameters (α, β) and the result was that the null hypothesis H_0 was accepted, which means that the value is closer to 0.2.

[13] `https://project.inria.fr/plasma-lab/examples/consensus-protocol` (visited on 2018-09-19)

Table 4.2: Dining philosophers CUSUM evaluation results with different initial numbers of philosophers.

Initial #Philosophers	p_{init}	k	Change detected at	
			Sample	#Philosophers
5	0.712	0.612	319	6
10	0.387	0.287	961	13
15	0.250	0.150	1072	18
20	0.157	0.057	1337	24

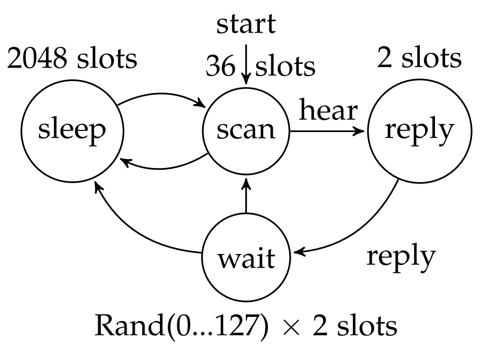


Figure 4.6: Bluetooth device discovery as presented for PRISM [60].

4.4.3 Bluetooth Case Study

We performed the third case study for a device discovery phase of Bluetooth, which is a wireless telecommunication standard [125]. This standard tries to avoid interference problems by applying a frequency hopping scheme. For this scheme, the devices use pseudo-random jumps between common sets of frequencies. Figure 4.6 illustrates the phases of the scheme.

It can be seen that there is a scan state, in which devices are listening for requests. When a device receives a request, then it enters a *reply* state, where it answers a request after two time slots. (A time slot has a duration of 312.5 μs.) Then, the device must wait for a random number of time slots. After this waiting time, the device goes back to the *scan* or the *sleep* state. In the *scan* state, a device can also start a *sleep* state to reduce the energy consumption, when no request was received. The case study was originally presented for PRISM [60] and later also for UPPAAL-SMC [56]. We based our implementation on the PRISM model. Compared to our previous case studies, the model was more complex, because it has a number of different modules, which interact through synchronisations. PRISM models support synchronised actions, which enable two or more modules to perform actions simultaneously. The model comprises modules for a sender, a receiver and for the frequency calculation. For our model implementation, we had to add functionality for the synchronisation. This was done by executing the corresponding actions on all modules when they were part of the synchronisation. We also had to make sure that the variable updates during these actions had no influence on the guards of the other executed actions. The rest of the implementation was similar to the one for the dining philosopher case study.

We checked the following properties:

1. What is the probability that we can observe k replies within a specified time?
2. What is the probability that the receiver sleeps at most s times until we observe k replies?

We performed a Monte Carlo simulation with 10000 samples to check the first property. The results are shown in Figure 4.7. It can be seen that the data points have a stair-like structure.

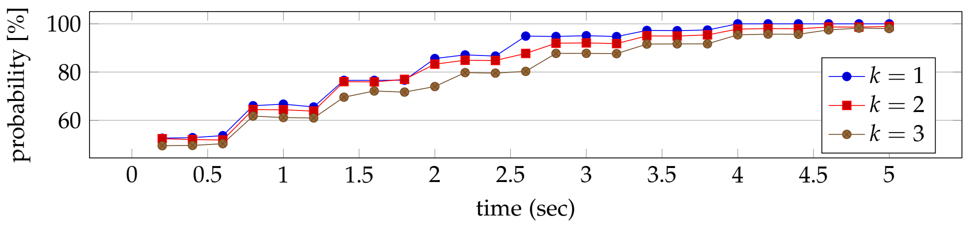


Figure 4.7: Bluetooth evaluation results for the property: what is the probability that we can observe k replies within a specified time?

Table 4.3: Bluetooth property: what is the probability that the receiver sleeps at most s times until we observe k replies?

Max Sleep	**Probability of Finishing [%]**		
Count s	$k = 1$	$k = 2$	$k = 3$
0	52.733	51.853	50.812
1	65.895	64.593	61.426
2	77.280	75.664	71.308
3	86.147	84.569	79.613
4	94.821	92.126	87.411
5	97.505	94.957	91.462
6	100.000	97.947	96.018
7	100.000	98.773	97.780
8	100.000	99.649	99.245

This is because of the sleep phases, which occur at certain probabilities and cause a sharp increase of the required time.

The second property was checked with a Monte Carlo simulation with Chernoff-Hoeffding bound with $\epsilon = 0.01$ and $\delta = 0.01$, which requires 26,492 samples. Table 4.3 shows the results for this property. We performed the evaluation until we observed k replies, and we checked in how many cases we can observe this number of replies before the receiver sleeps s times. As expected we see an increase of the probability of observing k replies, when the number of allowed sleep phases rises. The results we obtained for both properties were corresponding to the results of the case study from PRISM. Hence, our approach could also reproduce the simulation of a more complex stochastic model.

Furthermore, we performed an evaluation with the SPRT in order to check the property: is the probability that we can observe k replies within a certain time closer to x or y? We used the value 0.01 for α and β, and we checked the property for different time limits and values for x and y. The results are shown in Table 4.4. It can be seen that the alternative hypothesis was accepted in all cases for $k = 1$ and $k = 2$ and that the null hypothesis was always accepted for $k = 3$.

4.5 Concluding remarks

We have demonstrated that statistical model checking can quite easily be integrated into a PBT tool. We have implemented four commonly used SMC algorithms in the form of SMC properties and evaluated them on standard examples from the literature: the dining philosophers, a randomised consensus shared coin protocol and a Bluetooth device discovery protocol. The results are encouraging. The case studies revealed that our approach enables the definition of stochastic models and properties in a high-level programming language, which provides some benefits in the modelling style and should be easier to use for developers who are not familiar with (temporal) logics.

Table 4.4: Bluetooth property: is the probability that we can observe k replies within a certain time closer to x or y?

Time [s]	$H_0 : p_0 = x$	$H_1 : p_1 = y$	**Accepted Hypothesis**		
			$k = 1$	$k = 2$	$k = 3$
1	0.60	0.65	H_1	H_1	H_0
2	0.80	0.85	H_1	H_1	H_0
3	0.85	0.95	H_1	H_1	H_0

The elegance of our integration is due to the fact that our new SMC properties take a classical property to be checked as input parameter. This results in a very flexible SMC approach where, e.g., state-machine properties as well as algebraic properties can be checked.

Moreover, we demonstrated that our SMC properties also support conformance testing of implementations with stochastic failures against a correct model. This allows the assessment of the failure probability.

In the next chapters, we will demonstrate how we utilise this method for performance analyses.

5 Learning Response-Time Distributions for Extending Functional Models

This chapter is based on our publications at ICTSS 2017 [162], at QEST 2018 [3], at SETTA 2018 [12], and in the journal SQJO 2017 [9].

In this chapter, we illustrate how we learn response-time distributions that are applied for the extension of our functional models. First, we explain how we perform MBT to produce log data that contains response times of concurrent requests. With this data, we illustrate how we apply a linear regression to obtain response-time distributions that are integrated into our models. We demonstrate both these phases with our TFMS and MQTT case studies that were described in Section 1.4.2.

5.1 Model-Based Testing for Log Data

First, we perform model-based testing with PBT in order to produce log data that serves as a basis for learning response-time distributions, which will be explained in Section 5.2. This initial testing phase is conducted with a functional EFSM model. With this functional model, we perform classical PBT, which generates random command sequences that include test data. We run several testing processes concurrently in order to produce log data that includes response times of simultaneous system requests. Put differently, we perform load testing with concurrent PBT threads that interact with the system-under-test and log the resulting response times.

How the test-case generation works exactly and what properties we check was already discussed in Section 3.3.1. Now, we apply this approach to produce log data for the TFMS and for an MQTT broker implementation.

5.1.1 TFMS

As explained in Section 1.4.2, we investigate a web-service application called TFMS from our research projects that consists of various modules. One simplified model out of the Test Order Manager module that was introduced in Section 3.6.2 serves us as an example. The evaluation of the whole module will later be presented in more detail in Chapter 7. The example model supports tasks, like creating or editing Business Process Templates, which are objects of the application domain. These objects include attributes (form data) that are stored in a database and have to be set by the users. The model of this SUT is illustrated in Figure 5.1. This model is a hierarchical state machine, as explained in Section 3.4.2. There are sub-state machines for each Business Process Template object and *select* transitions can switch between these objects. We have a variable *activeObj* that identifies the currently open object and a map (*stateMap*) from object identifier to object state that stores the state of all objects. Each sub-state machine shows the tasks that can be performed for a Business Process Template object. In reality, each of these tasks represents a page of the system with required form fields (attributes). Hence, the transitions require attributes, which we omit for brevity. Note that the tasks of the sub-state machines in Figure 5.1 consist of multiple subtasks, e.g., for opening the page (*StartTask*), for setting attributes (*SetAttribute*), and for saving the page (*Commit*). Since most of these subtasks are requests to the web-server, we record them in our logs, which we will see later.

In Chapter 3, we have demonstrated how such functional models can be derived from business-rule models of the server implementation and how they can be applied for PBT. Now, we also utilise these models for PBT in order to generate random sequences of commands with

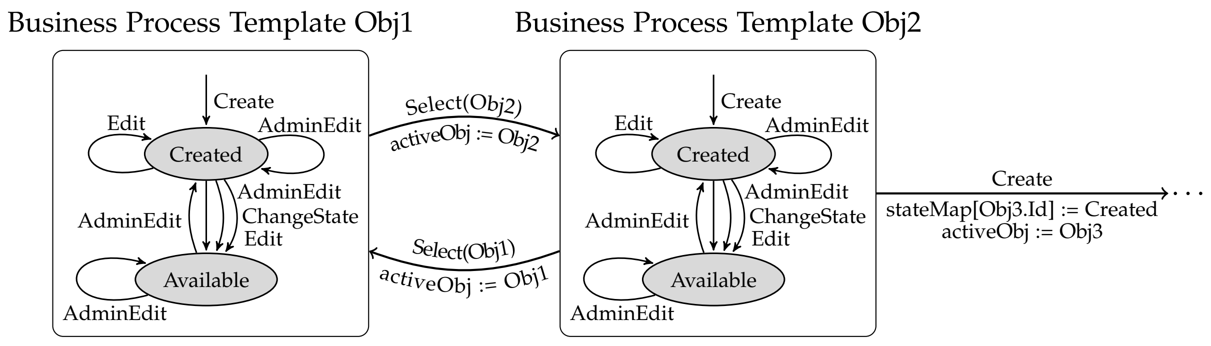


Figure 5.1: Functional EFSM model for Business Process Templates.

form data (attributes). In contrast to the previous chapter, our aim is to create log data that captures the response times of individual requests. The functional test of the SUT is only a positive side effect of this phase.

We run several testing processes concurrently on the SUT in order to obtain response times of multiple simultaneous requests. This represents the behaviour of multiple active users. An example log from a non-productive test system with low computing resources (virtual machine) is represented in Table 5.1. We record response times of tasks, subtasks, simultaneous requests (*#ActiveUsers*), the attribute name if the request considers only one attribute and otherwise the number of attributes (*#Attributes*), the generated form-data size (*ObjSize*), and the cumulative sum of the data size (*CumulativeObjSize*) (which represents the database fill level of the SUT). For this initial logging phase the test-case generator chooses the transitions, i.e., the tasks, with uniform distribution.

5.1.2 MQTT

We applied the method that we used to test the TFMS similarly for an MQTT broker implementation. The difference was that the broker did not have business-rule models. Hence, we created the functional model manually. Our model was inspired by learned models from Tappler et al. [173].

MQTT brokers allow clients to connect/disconnect, subscribe/unsubscribe to topics and publish messages for such topics. Each of these actions can be performed with a corresponding control message, which is defined by the MQTT standard [28]. We treat the broker as a black box and test it from a client's perspective.

The upper state machine in Figure 5.2 represents the messages that we test. We run multiple of these state machines concurrently in order to produce log data that includes latencies for simultaneous messages of several clients. Each transition of the state machine is labelled with an input i, an optional guard g / assignment operations op, and an output o. Some transition inputs are parametrised with generated data, e.g., a topic for *subscribe*. We apply PBT generators in order to produce inputs and their required data in the same way as explained

Table 5.1: Example log data of the TFMS Business Process Template model.

Task	Subtask	#Active-Users	Attribute	#Attributes	ObjSize	Cumulative-ObjSize	Response Time [ms]
Create	StartTask	5	-	-	-	7,115,966	21
Create	SetRefAttr	4	Responsible	-	0	7,119,938	17
ChangeState	StartTask	2	-	-	0	7,119,938	22
Create	Commit	5	-	5	3985	7,123,923	31
ChangeState	Commit	4	-	2	3372	7,181,842	25

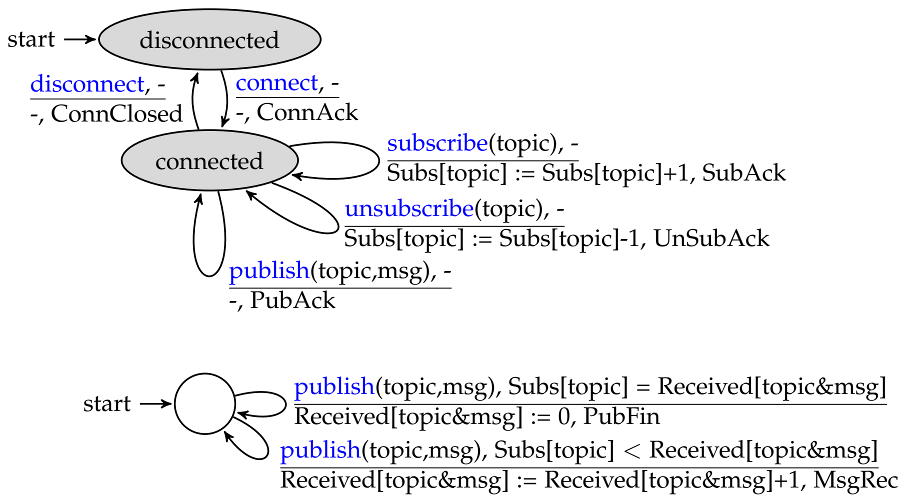

start
disconnected
disconnect, -
-, ConnClosed
connect, -
-, ConnAck
connected
subscribe(topic), -
Subs[topic] := Subs[topic]+1, SubAck
unsubscribe(topic), -
Subs[topic] := Subs[topic]-1, UnSubAck
publish(topic,msg), -
-, PubAck
start
publish(topic,msg), Subs[topic] = Received[topic&msg]
Received[topic&msg] := 0, PubFin
publish(topic,msg), Subs[topic] < Received[topic&msg]
Received[topic&msg] := Received[topic&msg]+1, MsgRec

Figure 5.2: Functional model for an MQTT client.

in Chapter 3. To keep it simple, we assume that a client can only subscribe to topics to which it did not subscribe before (the same for unsubscribe).

In order to manage the subscriptions, we have a global map *Subs* that stores the subscription numbers for each topic. This map is needed when publishing, because we want to check if the number of received messages corresponds to the number of subscribed clients. In order to perform this check, we have a second state machine (Figure 5.2 bottom) that represents the message receivers. This machine stores the number of received messages in a map *Received* that takes the topic concatenated with the message *(topic&msg)* as key. The map is updated for each message receiver, and when all messages were delivered, then a *PubFin* output is produced. For simplicity, we omit some assignment operations, e.g., for a subscriptions set.

Based on this functional model, we perform MBT with a PBT tool, which generates random test cases that are executed on an MQTT broker. During this testing phase, we capture the latencies of messages in a log file. Note that the latency is the duration that a client must wait until it receives a response to a sent message from the broker or until the message is delivered to all receivers in case of a *publish*. To simplify the discussion, we often only talk about response times, although we mean latencies and response times.

An example log excerpt from the MQTT implementation Mosquitto is presented in Table 5.2. It shows that we record the message type (*Msg*), the number of active clients or open message exchanges (*#ActiveMsgs*), the total number of subscriptions (*#TotalSubs*), the size of the topic (*TopicSize*) and message string (*MsgSize*), the number of subscribers for a topic when a *publish* occurs (*#Subs*), the number of receivers of a published message (*#Receivers*), and the latency. For this initial logging phase, the available transitions in the current state of the functional model (Figure 5.2 top) are chosen with a uniform distribution. In the *disconnected* state, the only choice is a *connect* message and in the *connected* state all other messages are selected with equal frequency. We do not apply any sojourn times between sending messages in this phase, since we want to capture the latencies in situations with many concurrent messages.

Table 5.2: Example log data of the MQTT broker Mosquitto.

Msg	#ActiveMsgs	#TotalSubs	TopicSize	MsgSize	#Subs	#Receivers	Latency [ms]
connect	47	266	0	-	-	-	110.82
subscribe	47	270	14	-	-	-	2.45
publish	47	270	14	52	7	7	32.72
unsubscribe	45	12	14	-	-	-	1.25
publish	46	272	14	74	1	1	2.13

5.2 Learning Response-Time Distributions with Linear Regression

For our case studies, we did not have the possibility to obtain log data from real users. The reasons were that (1) it was not feasible to obtain permission from TFMS customers to use their data from a productive environment, (2) for MQTT it would have been necessary to build a huge network with various real devices in order to form an interesting setup, and (3) the time for the log data generation would have been too high in such realistic conditions where the usage frequency is typically low.

Hence, we applied an MBT approach that is based on PBT as explained in the previous section in order to produce log data. The advantages of this approach are that we can run it on demand, with various settings that may be needed, and we have high flexibility, e.g., we can record all kinds of attributes. However, a disadvantage of this approach is that our generated logs might be biased. For example, a bias might be caused by log data generation that is not random enough. In this case, we could obtain an artificial correlation between variables (or features). We had to be careful with our test setup in order to avoid such biases [53].

Furthermore, the measurements might be perturbed by: (1) network delays and interruptions, or hardware utilisation influences when test clients run on the same machine as the SUT, (2) the run time of the testing tool for the log data generation. We tried to avoid these issues by performing the experiments on a machine with sufficient system resources so that there are no shortages, and the network was selected fast enough to reduce the influence of network aspects, like delays. Moreover, we made sure that the time overhead of our testing tool was negligible.

Note that we also had to keep in mind the following requirements for our learning method:

1. A prediction should be fast enough, i.e., it must not take more than a fraction of the predicted response time. This is necessary, because the prediction is needed to simulate the requests to SUT with a virtual time that is a fraction of real time. More specifically, our simulation would not make sense, if the prediction took longer than actually performing the requests to the SUT, because then we could just execute the SUT instead of using a prediction model.
2. The integration of the prediction model into our testing tool should be simple, since we want to apply this tool for both simulating the expected response times of the SUT and testing the simulation result on the real system.

For learning response times with multiple linear regression (Section 2.3) [152], we apply a process that consists of several phases, i.e., data cleaning and pre-processing, feature selection, model evaluation, and model integration. These phases are described in the following subsection.

5.2.1 Data Cleaning and Pre-Processing.

In this phase, we check for biases and perform data cleaning, i.e., we remove invalid or problematic log entries [101].

A bias could be that the data generation might not be random enough or that it might unintentionally be set up in a way, where relevant scenarios for the prediction were not tested frequently enough. Both these issues can result in a regression model that has a good performance for the training data, but it would not produce reliable predictions for our simulation with SMC. In order to reduce the risk of biases, it is helpful to carefully analyse the data with visualisations, e.g., with scatter plots, histograms or correlation matrices. For example, if a correlation matrix shows correlations that should not be there, then this might indicate a problem during the initial test-case generation.

For the *data cleaning*, data visualisations also helped to find issues with the data. In this phase, we performed the following steps.

- We remove entries with long response times, i.e., outliers. For example, we are not interested in unusually long response times that are caused by network disruptions. Our aim is to maximise the user satisfaction. Hence, we are mainly interested in average response times under normal conditions, but not in worst-case scenarios. We consider the top 5% of the entries per message or request type as outliers.
- We skip entries of the log files that are generated during the first minutes of the testing process, because there are various initialisation steps, like a cache setup, which lead to unrepresentative data at the beginning.
- Moreover, we flag and remove entries where exceptions were raised, e.g., due to timeouts or connection failures, since they are rare, and we are primarily interested in successful requests or message exchanges.
- We restart the testing phase, when the data indicates that there was a crash, e.g., when a testing process of a user stopped unexpectedly.
- Finally, we set missing values for required request attributes to a standard value, e.g., for categorical attributes we set them to *NOTSET*, which represents an additional category.

5.2.2 Feature Selection.

Next, we applied *feature selection*, where we select variables that have a significant influence on the target variable, i.e. the response-time or latency in our log data [78].

In this phase, it is important to have a good understanding of the SUT and the corresponding data in order to select features that lead to an accurate model [172]. We can again inspect data visualisation to facilitate this phase. For example, we can apply correlation matrices and look for features that are correlated with the target variable. The correlation can be measured with a correlation coefficient r, e.g., a common one was introduced by Pearson [145] and gives us a value $r \in [-1, 1]$, where 1 is a total positive correlation and -1 a negative correlation. Features that have a medium or strong correlation $r \geq 0.3$ are most important for the regression, but sometimes also features with a weak correlation $0.1 < r \leq 0.3$ can help to improve the regression model.

Table 5.1 and Table 5.2 show the attributes that we recorded during the data-generation phase, since we found that they are important for learning the response times. However, we performed several feature-engineering steps in order to build a better prediction model.

- We checked if certain features only have an effect on specific request or message types. For example, we noticed that the database fill level only has an influence on specific requests of the TFMS. Multiplying this feature with a Boolean variable (evaluating to one in case that it has an influence and zero otherwise) allowed us to enable or disable this feature for specific requests. This helped to further improve the performance of the model. We also applied this approach for the *#ActiveMsgs* variable of MQTT.
- We combine features in order to form a new combined feature, when certain features belong together and when this shows an improvement. For example, we combine the features *Task* and *Subtask* to form a new feature, called *Task_Subtask*, because we observed that the same subtask can have a completely different behaviour depending on the corresponding task.
- We avoid features that have a high correlation among each other, since they might be redundant. For example, the number of subscribers to a topic of a published message is highly correlated with the number of message receivers. Hence, we only select one of these.
- We duplicate a feature when we notice that it has different correlations depending on the request or message type. Thereby, we can enable or disable one copy of this feature for a specific type in order to represent the different correlations.

For the TFMS, we selected all features that were presented in Table 5.1 for our regression model, and we applied the mentioned feature-engineering steps. For the MQTT case study, we selected only a subset of the attributes of Table 5.2 and applied the following regression formula:

$$\mathit{Latency} \sim \mathit{Msg} + \#\mathit{ActiveMsgs} + \#\mathit{TotalSubs} + \#\mathit{Subs}$$

Note that categorical variables, like *Msg*, cannot be directly used for the regression. They need to be encoded first. This can, e.g., be done with a dummy coding, where each category is represented by a binary dummy variable that is set to one if the record has this category and to zero otherwise [152].

We performed multiple linear regression as explained in Section 2.3 to learn a prediction model that produces response-time distributions for our given log data. This algorithm was able to fulfil the requirements of our learning method that were mentioned earlier and still produced predictions that worked well enough for our method, as we will later see in Chapter 7.

The well-known statistic tool *R* with the standard *lm* function was applied to produce the regression model for MQTT.[14] For the TFMS, a custom tool was implemented by Cristinel Mateis that facilitates the described data cleaning and pre-processing steps. This tool was developed in the programming language *python* 2.7, and it applies the *scikit-learn*[15] 0.19.1 machine-learning package in order to produce a prediction model.

Next, we show how to measure the prediction power of our generated regression models.

5.2.3 Model Evaluation.

The data that is taken as input for a learning algorithm is called the training data. Although our model might work well with this training data, we cannot be sure if it will also be reliable for new data. Hence, we apply a method called k-fold cross validation [78] that allows us to estimate how our model will work for unseen data. For this method, we randomly split the training data into k subsets of equal size. Then, we perform our learning algorithm by only taking $k-1$ of these subsets as training data. The last subset is applied to check, how our model will perform on unseen data. We repeat these steps k times and in each iteration we omit a different subset from the selected training data. When the prediction of a model is reliable in all iterations, then we can be confident that our model did not work well just by chance, and it is likely that it will also predict well for unseen data.

The quality of the regression model can be measured with the coefficient of determination (R^2-score) [133, 194], which describes how well a regression model fits given data. This score can be calculated as follows:

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}, \tag{5.1}$$

where N is the number of log entries, y_i is the response time of the i^{th} entry, $\hat{y}_i$ is the predicted response time of the model for the attributes of the i^{th} log entry, and $\bar{y}$ is the mean of all N response times of our log data. The value of $R^2 \in [0,1]$ is high when the model produces good predictions and the value 1 represents a perfect prediction.

Our obtained R^2-scores were in the range $[0.7, 0.95]$, depending on the selected setup of the initial testing phase. We also computed the R^2-scores for the k-fold cross-validation. The validation was performed with $k = 5$ and had comparable R^2-scores for all iterations of this method. This gives us confidence that we have no overfitted model, which could have a high R^2-score, but would perform bad for unseen data.

[14] `https://www.r-project.org` (visited on 2018-09-19)
[15] `http://scikit-learn.org` (visited on 2018-09-19)

```
                                Estimate     Std. Error   t value      Pr(>|t|)
(Intercept)                      26.2752      0.1714      153.2138     0.0
#Users                            4.4268      0.0149      297.0559     0.0
#Attributes                       0.8998      2.1773        0.4132     0.6793
ObjectSize                        0.0023      0.0028        0.8253     0.4091
CumulativeObjSize                 1.2698e−06  4.3829e−09  289.7210     0.0
AdminEdit_SetRefAttribute         4.9903      0.1359       36.7000     4.9008e−294
AdminEdit_StartTask               0.8427      0.5624        1.4985     0.1340
Create_Commit                    −7.9862      0.2186      −36.5286     2.5241e−291
Create_SetRefAttribute            4.5492      0.1452       31.3174     7.4209e−215
Create_StartTask                  0.6055      0.5650        1.0716     0.2838
ChangeState_Commit               −4.6380      4.6892       −0.9890     0.3226
ChangeState_SetRefAttribute       4.6955      0.2157       21.7593     7.1371e−105
ChangeState_StartTask             1.0689      0.5769        1.8528     0.0638
Edit_Commit                      −2.7279      0.3268       −8.3468     7.0478e−17
Edit_SetRefAttribute              5.2434      0.3042       17.2366     1.5486e−66
Edit_StartTask                    4.1145      0.6213        6.6217     3.5560e−11
Attribute_NOTSET                −15.4171      0.5172      −29.8043     7.8762e−195
Attribute_Responsible           −18.6690      0.2332      −80.0309     0.0
...
```

Listing 5.1: Regression model excerpt for TFMS Business Process Templates.

Note that we also tested other algorithms, like a polynomial regression or random forests, but they only achieved slightly better R^2-scores and have a higher complexity. Hence, we still kept using the multiple linear regression, since it is a simple method that allows an easy integration in our simulation environment. However, the investigation of further learning algorithms is still a promising topic for future work.

5.2.4 Integration of the Response-Time Distributions.

Listing 5.1 illustrates a regression model that was generated for our TFMS example. The left column shows the intercept and the regressors $[x_0, \ldots, x_p]$ from the model (2.8), which was explained in Section 2.3. Note that for categorical variables, like *Task_Subtask*, we have multiple entries. The second column lists the estimates for the means (μ_{β_k}) and the third shows the standard errors of these estimates (σ_{β_k}) that represent the average variation of an estimate from the actual mean value. The fourth column are the t-values that show the ratio of the estimate and the standard error, and the last column are p-values that indicate the significance of our estimates, i.e., a small p-value shows a high significance. These p-values can be applied to simplify the model, which can be done by omitting features with a high p-value.

For MQTT, the regression model has a similar form as the one for the TFMS, but in contrast to the TFMS it was produced with R. The regression was performed with log data from Mosquitto, which contained 100 test cases with a random number of clients (3–100) and a length of 50 messages. This produced log files with about 300,000 entries. The required number of test cases was determined by increasing the dataset size in a stepwise manner and by executing the regression, until the R^2-score did not increase any further. (For the TFMS, we had about two million entries in our logs, due to the higher complexity.)

Listing 5.2 shows the results of the multiple linear regression for MQTT. The columns are the same as in the previous model. Note that the label $***$ at the end of each line shows that the variables are all highly significant.

In order to use this regression model in our method for MQTT, we encode it in a *latency* function that takes the message type, the number of active messages, the total number of

```
1                          Estimate  Std.Error  t value   Pr(>|t|)
2  (Intercept)            −8.009707  0.1106356  −72.397  < 2e−16 ***
3  Msgdisconnect           8.084679  0.1234019   65.515  < 2e−16 ***
4  Msgpublish              9.066681  0.1395017   64.993  < 2e−16 ***
5  Msgsubscribe            8.771242  0.1419899   61.774  < 2e−16 ***
6  Msgunsubscribe          9.294850  0.1294843   71.784  < 2e−16 ***
7  #ActiveMsgs             1.358794  0.0033433  406.417  < 2e−16 ***
8  #TotalSubs              0.002503  0.0002084   12.011  < 2e−16 ***
9  #Subs                   0.294270  0.0307663    9.565  < 2e−16 ***
```

Listing 5.2: Linear regression output (excerpt) for the MQTT broker Mosquitto.

subscribers, and the number of subscribers for the currently published message as input and returns the parameters μ_y and σ_y of the normal distribution of the latency as result:

$$latency : Msg \times \mathbb{N}_{>0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \to \mathbb{R} \times \mathbb{R}$$

For the TFMS, we have a similar function called *rtime*:

$$rtime : Task \times Subtask \times \mathbb{N}_{>0} \times Attribute \times \mathbb{N}_{>0} \times \mathbb{N}_{>0} \times \mathbb{N}_{>0} \to \mathbb{R} \times \mathbb{R}$$

This function takes a task, a subtask, the number of active users, an (optional) attribute, the number of attributes, the size of the generated form data for the request, and the cumulative sum of the generated data sizes as input and returns the parameters of a normal distribution for the response time.

In both these functions, we perform a linear combination of the distributions given by the estimates and standard errors of the associated regression coefficients for the inputs. This gives us a combined normal distribution that depends on the inputs of this function. We apply the formulas (2.8) from Section 2.3, with the input parameters $[x_0, \ldots, x_p]$ of these functions, and the parameters $(\mu_{\beta_k}, \sigma_{\beta_k})$ of the regression model. As a result we obtain the mean μ_y and the standard deviation σ_y of the normal distribution.

For example, when we apply our regression model for MQTT (Listing 5.2) and consider a *subscribe* message that happens when 15 other messages are active and when there are zero subscribers, the linear combination works as follows. The associated regression coefficients (Lines 2, 5 & 7 in Listing 5.2) are combined in order to obtain parameters for a normal distribution:

$$\mu_y = -8.010 + 8.771 + 15 \times 1.359 \qquad \sigma_y = \sqrt{0.111^2 + 0.142^2 + (15 \times 0.003)^2}$$

In the next chapter, we show how we integrate these functions into our functional models, and we combine these models with usage profiles in order to form combined models that we can evaluate with SMC.

Note that in our previous work on TFMS [162], we named the distribution as cost distributions, because we wanted to highlight that our approach may be generalised for predicting other types of performance indicators, like the energy consumption. In this thesis, we explicitly talk about response time in order to avoid confusions.

6 Statistical Model Checking for Predicting and Testing Response-Times

This chapter contains parts of our publications at ICTSS 2017 [162], at QEST 2018 [3], at SETTA 2018 [12], and in the journal SQJO 2017 [9].

In this chapter, we demonstrate the simulation of usage profiles by integrating them into our functional model. Additionally, we add the learned response-time distributions that were presented in the previous chapter. Based on this combined model, we illustrate a prediction method that computes probability results of queries about the expected response time of users. Moreover, we introduce a hypothesis-testing technique that allows us to evaluate such predictions directly on the SUT. Finally, we discuss the implementation of our method that was realised via PBT.

6.1 Monte Carlo Simulation of the Model for Predicting Response Times

In order to apply SMC for a realistic usage scenario, we integrate given usage profiles and response-time distributions derived using linear regression into the functional models that were explained in Section 5.1. An example usage profile for the Business Process Template model that was introduced in the previous chapter is shown in Listing 6.1. The usage profile is encoded in the JavaScript Object Notation (JSON) format. It includes weights for tasks, user input (waiting) time intervals between tasks/subtasks that represent the time that a user needs for the input and data specific waiting factors, e.g., a delay per character, or a delay per reference for the number of options of a drop-down menu.

For MQTT, we also a have usage profile (UP1) in a similar form, which is shown in Listing 6.2. This profile describes the behaviour of an MQTT client, i.e., how long it should wait between sending messages, and with what probabilities it should send certain messages. The time between messages is selected uniformly inside the bounds *[MinTimeBetwMsg, MaxTimeBetwMsg]*, and we have weights that define the message frequency.

The extension of the initial functional model with a usage profile and response-time distributions gives us a combined model that is a stochastic timed automaton as explained in Section 2.4. Figure 6.1 illustrates such an automaton for our TFMS example. Note, we only show the combined model of one sub-state machine of the hierarchical EFSM in Figure 5.1 for brevity. All locations (states) l in this combined model include a sojourn time that is defined with a probability density function f_l. The tasks of the functional model where separated into subtasks in order to represent the response times of individual requests. Each subtask comprises an edge that calls the *rtime* function to receive the parameters (μ, σ) and a location (d_i) that defines f_{d_i} as a normal distribution with these parameters. All other locations define f_l as a uniform distribution given by an upper and lower bound $[a, b]$. The locations *Submitted* and *Closed* have bounds from the user input time intervals between tasks of the usage profile and for the other locations (w_i), the bounds are calculated in a separate edge with a function

```
{ TaskWeights: { Create: 35, Edit: 65, AdminEdit: 1, ChangeState: 1, Select: 5 },
  TaskWaitIntervalStart: 500, TaskWaitIntervalEnd: 1500, SubTaskWaitIntervalStart: 300,
  SubTaskWaitIntervalEnd: 500, WaitPerReference: 10, WaitPerCharacter: 30 }
```

Listing 6.1: Usage profile for the TFMS Business Process Template model.

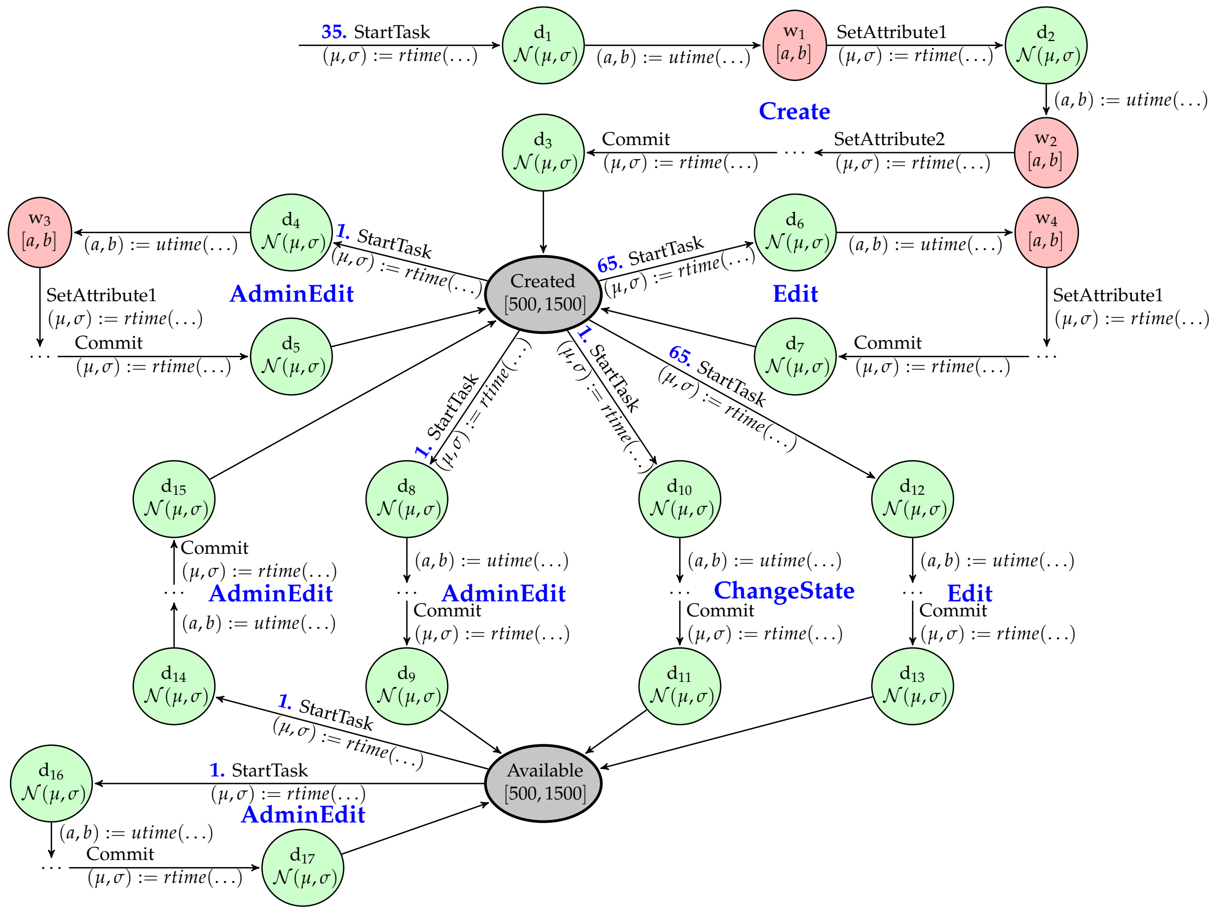

35. StartTask
(μ,σ) := rtime(...)
d1 N(μ,σ)
(a,b) := utime(...)
w1 [a,b]
SetAttribute1 (μ,σ) := rtime(...)
d2 N(μ,σ)
(a,b) := utime(...)
Create
d3 N(μ,σ)
Commit (μ,σ) := rtime(...)
SetAttribute2 (μ,σ) := rtime(...)
w2 [a,b]
w3 [a,b]
(a,b) := utime(...)
d4 N(μ,σ)
1. StartTask (μ,σ) := rtime(...)
Created [500,1500]
65. StartTask (μ,σ) := rtime(...)
d6 N(μ,σ)
(a,b) := utime(...)
w4 [a,b]
SetAttribute1 (μ,σ) := rtime(...)
AdminEdit
Edit
Commit (μ,σ) := rtime(...)
d5 N(μ,σ)
d7 N(μ,σ)
Commit (μ,σ) := rtime(...)
1. StartTask (μ,σ) := rtime(...)
1. StartTask (μ,σ) := rtime(...)
65. StartTask (μ,σ) := rtime(...)
d15 N(μ,σ)
d8 N(μ,σ)
d10 N(μ,σ)
d12 N(μ,σ)
Commit (μ,σ) := rtime(...)
AdminEdit
(a,b) := utime(...)
(a,b) := utime(...)
AdminEdit
Commit (μ,σ) := rtime(...)
(a,b) := utime(...)
ChangeState
Commit (μ,σ) := rtime(...)
(a,b) := utime(...)
Edit
Commit (μ,σ) := rtime(...)
d14 N(μ,σ)
d9 N(μ,σ)
d11 N(μ,σ)
d13 N(μ,σ)
1. StartTask (μ,σ) := rtime(...)
d16 N(μ,σ)
1. StartTask (μ,σ) := rtime(...)
AdminEdit
Available [500,1500]
(a,b) := utime(...)
Commit (μ,σ) := rtime(...)
d17 N(μ,σ)

Figure 6.1: Stochastic timed automaton of one TFMS Business Process Template object.

utime (user time). This function takes into account the user-time intervals between subtasks and the data-dependent time, e.g., the delay per character, from the usage profile, and returns according bounds. The task weights of the usage profile are attached to the edge weights w_e and they are shown before an edge name (in bold). It can be seen that each transition or task of the initial functional model is now represented as a sequence of edges with a silent edge at the end. Note that the *Create* and *Select* tasks are also possible in the *Created* and *Available* location, but we omit additional edges for these in order to keep the figure simple. We also omit parameters and their assignments for the *rtime* and *utime* functions. The parameters for *rtime* were already explained before and *utime* takes the generated attribute data as input and returns associated intervals for the user-input time.

A similar extension is applied to the functional MQTT model that is shown in the top of Fig 5.2. For this purpose, latency distributions and our usage profile (Listing 6.2) are integrated into the functional MQTT model. As a result, we also obtain a combined timed model in the form of a stochastic timed automaton, as illustrated in Figure 6.2. It can be seen that the *connected* and *disconnected* locations have a uniform distribution given by an upper and lower bound $[a,b]$. These bounds come from our usage profile. All other locations have a normal distribution for the sojourn time. The parameters for this distribution are computed by

```
{ MinTimeBetwMsg: 0, MaxTimeBetwMsg: 500,
MsgWeights: { connect: 1, disconnect: 1, publish: 5, subscribe: 3, unsubscribe: 2 } }
```

Listing 6.2: MQTT usage profile UP1 with time bounds and weights for messages.

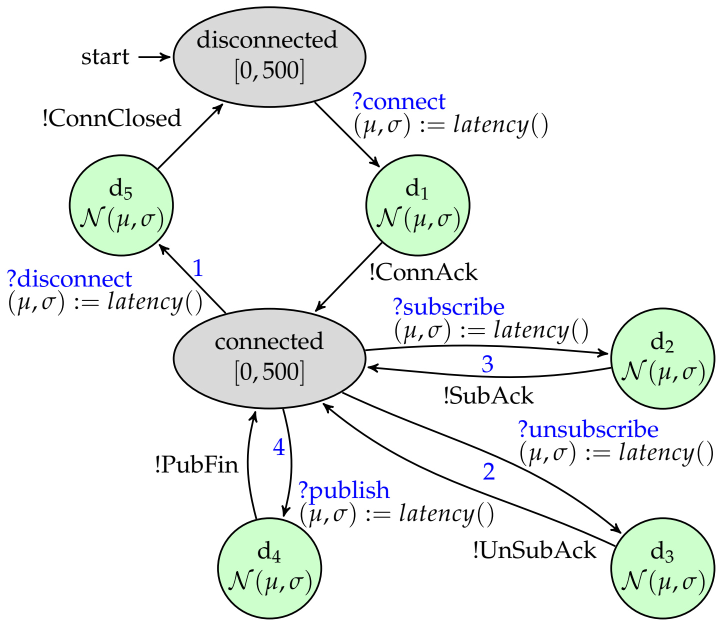

start
disconnected
[0, 500]
?connect
$(\mu, \sigma) := latency()$
d1
$\mathcal{N}(\mu, \sigma)$
!ConnAck
!ConnClosed
d5
$\mathcal{N}(\mu, \sigma)$
?disconnect
$(\mu, \sigma) := latency()$
1
connected
[0, 500]
?subscribe
$(\mu, \sigma) := latency()$
3
!SubAck
d2
$\mathcal{N}(\mu, \sigma)$
?unsubscribe
$(\mu, \sigma) := latency()$
2
!UnSubAck
d3
$\mathcal{N}(\mu, \sigma)$
4
!PubFin
?publish
$(\mu, \sigma) := latency()$
d4
$\mathcal{N}(\mu, \sigma)$

Figure 6.2: Stochastic timed automaton for the timing behaviour of an MQTT client.

the *latency* function introduced in the previous chapter. In contrast to the functional model, we have additional locations that apply the message latencies. These locations have one incoming edge that represents sending a message and an outgoing edge for the response. Moreover, the weights w_e from our usage profile are added to the transitions for sending messages. Note that we have omitted the parameters of the *latency* function and also some assignments that are necessary for these parameters, in order to keep the figure more readable.

With such combined stochastic timed automata models, we can evaluate usage profiles by simulating the expected response times or latencies. Furthermore, we can analyse a user population or setup consisting of multiple users or clients, by running several models concurrently. While we execute the model, we can check properties to answer questions, like "What is the probability that the response time of all requests within a task sequence of a fixed length, i.e., a test case, is under a specific threshold for each user within a population?", or "What is the probability that the latency of each interaction of a client within a given MQTT setup is under a certain threshold?".

Such questions can be answered with a Monte Carlo simulation with Chernoff-Hoeffding bound as explained in Section 2.2.2. For example, predicting the probability that the response time of all subtasks is under a threshold of 50 ms for each user of a population of 20 users with parameters $\epsilon = 0.05$ and $\delta = 0.01$, requires 1060 samples and returns a probability of 0.806, when a test-case length of four tasks is considered for the TFMS example.

Checking the probability that a latency threshold of 50 ms is satisfied for each client of an MQTT setup with 130 clients with the same parameters ($\epsilon = 0.05$, $\delta = 0.01$) requires also 1060 samples, and returns a probability of 0.84, when a test-case length of ten is considered.

Such evaluations require too many samples to be efficiently executed on the SUT, and hence, we only run them on the model. Fortunately, the sequential probability ratio rest (SPRT) requires fewer samples, and is therefore better suited for evaluating the SUT as we will see in the next section.

6.2 Hypothesis Testing of the System-Under-Test for Checking Response-Time Predictions

The SPRT, which was described in Section 2.2.3, is a form of hypothesis testing. It allows to check if the probability of a property is closer to a probability that is defined by a null hypothesis or to one that is defined by an alternative hypothesis. We apply this algorithm in order to check our predictions that were computed with the model as explained in the previous section. The predicted probabilities serve as hypotheses for testing the performance of real systems.

For example, we perform three different types of evaluations in order to assess the probability of a property about the expected response time of the SUT.

Evaluation 1. We check if the SUT is at least as good as our model predicted, i.e., if we can observe a probability on the SUT that is greater or approximately equal to the predicted probability of the model.

Evaluation 2. We test the prediction power of our model by checking if the probability of the SUT is only marginally lower or higher than the predicted probability of the model. In other words, we check if the SUT is at least as good as our model predicted and additionally we test if the SUT is not much better.

Evaluation 3. We evaluate if the probability prediction of a reference SUT is also observable on SUT deployments or configurations that have a different hardware or network setup. More specifically, the probability that was computed on the model serves as a hypothesis in order to check, if other SUT deployments are at least as good as the reference SUT.

For all these evaluations, we select the probability that was computed with the model p_m as alternative hypothesis H_1. Then, for Evaluation 1, we select a null hypothesis that is smaller (by a value Δ), because we want to be able to reject the hypothesis that the SUT has a smaller probability. The hypotheses can be defined similar as described in Section 2.2.3 and look like this:

$$H_0 : \theta = \theta_0 \mid f(x_i, \theta_0) = \begin{cases} 1 - p_m - \Delta & \text{if } x_i = 0 \\ p_m - \Delta & \text{if } x_i = 1 \end{cases} \qquad H_1 : \theta = \theta_1 \mid f(x_i, \theta_1) = \begin{cases} 1 - p_m & \text{if } x_i = 0 \\ p_m & \text{if } x_i = 1 \end{cases}$$

Evaluation 2 works similar, but includes an additional hypothesis test with a larger null hypotheses, which allows us to reject the hypothesis that the SUT has a higher probability than the predicted one from the model. This additional hypothesis test has the following hypotheses:

$$H_0 : \theta = \theta_0 \mid f(x_i, \theta_0) = \begin{cases} 1 - p_m + \Delta & \text{if } x_i = 0 \\ p_m + \Delta & \text{if } x_i = 1 \end{cases} \qquad H_1 : \theta = \theta_1 \mid f(x_i, \theta_1) = \begin{cases} 1 - p_m & \text{if } x_i = 0 \\ p_m & \text{if } x_i = 1 \end{cases}$$

Moreover, also Evaluation 3 is similar to the Evaluation 1, with the only difference that we test other SUT deployments. In this case, the model is derived from data from a reference SUT, and we want to know if a comparable performance can be achieved, when the system is deployed on a different hardware or network setup. The data flow of the overall process of this evaluation is illustrated in Figure 6.3.

1. We perform MBT with a functional model and capture the response times of requests of a reference SUT as log data as described in Section 5.1.
2. The log files are then taken as input for a linear regression, which gives us response-time distributions (see Section 5.2).

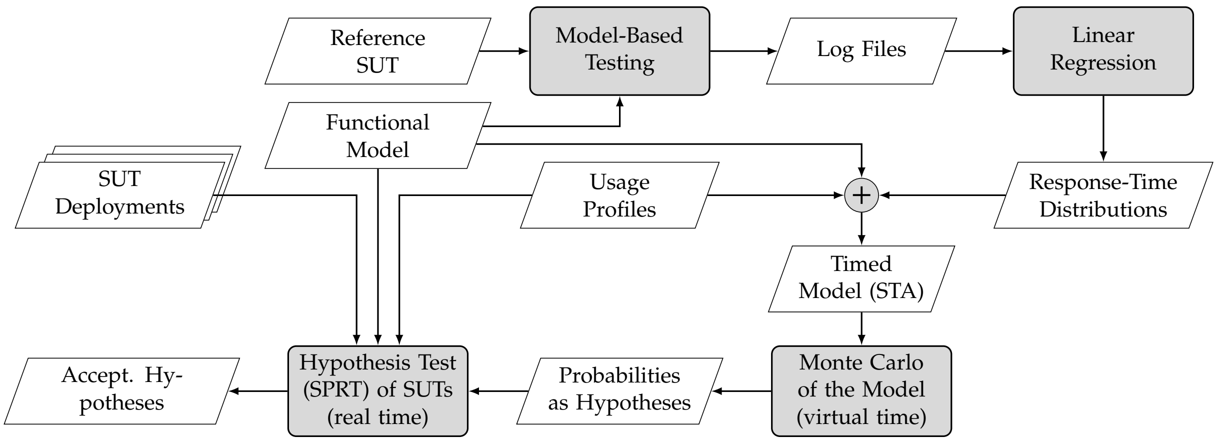


Figure 6.3: Overview of the data flow of our deployment-testing method.

3. The distributions and stochastic usage profiles are integrated into the functional model, resulting in a combined stochastic timed automata (STA) model.
4. Next, we perform a Monte Carlo simulation of this model to answer queries about the expected response time of users as demonstrated in the previous section.
5. Finally, the resulting probabilities serve us as hypotheses in order to check if deployments of the SUT can satisfy the same response-time thresholds as the reference system.

Note that in order to obtain an average number of needed samples, we run the SPRT concurrently for each user or client of a specified population and calculate the average of these runs. Multiple independent SPRT runs would produce a better average, but these tests have a high computation time, and we only have limited time in the test environment. We apply the SPRT algorithm with 0.01 as type I and II error parameters (α and β) in all the following examples.

Evaluation 1 was performed with our MQTT example, where we consider the predicted probability 0.84 of the model as alternative hypothesis and select a probability of 0.74 as null hypothesis, which is 0.1 smaller. As a result, the alternative hypothesis (probability 0.84) was accepted for all clients and on average 41.15 samples (test cases) were needed for the decision. The acceptance of the alternative hypothesis means that the SUT was at least as good as the model predicted in this case.

We applied Evaluation 2 to our TFMS example. The computed probability of the model (0.806) served as alternative hypothesis, and we select a probability of 0.556 as null hypothesis, which is 0.25 smaller. Additionally, we tested a probability of 1, which is about 0.2 larger than the computed probability, as a null hypothesis with a second SPRT and the same alternative hypotheses. The alternative hypotheses were accepted for both SPRTs and for all users, which means that the model's prediction was accurate, and on average only 17.55 and 11 samples were needed for the first and second SPRTs, respectively.

Moreover, we applied Evaluation 3 (deployment testing) to the TFMS. After we have evaluated the hypotheses on the reference SUT, we can reuse the hypotheses to check if different deployments of this SUT provide a similar performance. For example, we reused the hypotheses that were shown for Evaluation 2 (a probability of 0.806 as alternative hypothesis and of 0.556 as null hypothesis) and applied them to a deployment that had 4 GB instead of 15 GB that were installed in the reference SUT. The result was the same as for the reference system, i.e., the alternative hypothesis was accepted by all clients, and about the same number of samples was needed.

The acceptance of the same hypotheses means that the deployment provides the same or a similar performance as the reference SUT for our specific usage scenario, otherwise the deployment has worse response time. Detailed results with different deployment evaluations follow in Section 7.2.

Note that the selected difference for the null hypothesis has an effect on the number of samples, i.e., the smaller the difference the more samples are usually required. We selected a difference of 0.25 for the TFMS, since we wanted to detect a noticeable difference from the user's perspective. For MQTT, we work with a smaller difference of 0.1, because we wanted to detect differences of MQTT implementations with similar performance, which we will see in the next chapter.

In summary, it can be said that the SPRT allowed us check properties with a practicable number of samples, which is especially important when the test-case execution on the SUT is costly. The examples showed that we only needed a fraction of the samples compared to the 1060 samples that were required for the Monte Carlo simulation with Chernoff-Hoeffding bound.

In the next chapter, we will see an extensive evaluation of our method, but first we discuss the implementation in the following section.

6.3 Implementation of the Response-Time Prediction and Testing Method

In this section, we illustrate how our timed models can be executed with PBT. We introduce custom generators for the simulation of response times and also for latencies, which work similarly for the user-input times of our usage profiles. Moreover, we illustrate how they are applied to generate test cases. Note that we released the source code of our implementation for MQTT in order to make our method available to the public.[16]

In Chapter 3, we already presented an implementation for model-based testing with FsCheck, which supports automatic form-data generation and EFSMs. In this previous implementation, we had command instances for transitions and generators for different data types (e.g., for form data).

Based on this existing implementation, we developed the following extensions in order to support our new method. The first extension is a parser that reads the learned response-time distributions and integrates them into the model. In the previous implementation, we had command instances, which represent the tasks and generators for different data types (of form data). Now, we introduce response-time generators for the simulation of requests, which can be applied in the same way as normal generators for test data. During the test-case generation, the generated response times can be evaluated within the commands, which is useful for our response-time analyses.

Algorithm 13 represents the implementation of a response-time generator that we applied for the evaluation of the TFMS. The inputs are a task, a subtask, an attribute (for requests that are only concerned with one attribute), an array of encapsulated attributes (for requests that transfer multiple attributes), and the *rtime* function, which returns the parameters μ and σ of the normal distribution and was introduced in the previous chapter. Additionally, there are global variables *ActiveUserNum* for the number of active users and *CumulativeObjSize* for the representation of the database size, which are shared by all users. The generator is expressed as a function that is called during the generation process and it works as follows. First, a sequence generator is applied to generate values for the *encapsulatedAttr* array, and the *select* function further processes the generated values and constructs a new generator that is then returned. *select* was already explained in Section 3.5. This function takes an anonymous function, which takes the value of the original generator as input, and returns a new value

[16]`https://github.com/schumi42/mqttCheck` (visited on 2018-09-19)

Algorithm 13 Pseudo code of a response-time generator for the TFMS.

```
Inputs: Task $t$,
        Subtask $st$,
        Attribute $a$,   ▷ this variable is needed for requests that are only concerned with one attribute
        Attribute[ ] $encapsulatedAttr$,   ▷ attribute array for requests that transfer multiple attributes
        $rtime : (\ldots) \rightarrow (\mu, \sigma)$   ▷ function for the simulation of the response-time
Global Variable: $ActiveUserNum \in \mathbb{N}$,   ▷ number of users that have an open request
                 $CumulativeObjSize \in \mathbb{N}$   ▷ sum of the data sizes of the created objects
1: function Generator
2:     return $Gen.sequence(encapsulatedAttr).select(data \rightarrow \{$
3:         $ActiveUserNum \leftarrow ActiveUserNum + 1$   ▷ should be locked (Mutex)
4:         $delay \leftarrow sample(rtime(t, st, ActiveUserNum, a, encapsulatedAttr.length,$
               $sizeOf(data), CumulativeObjSize))$   ▷ sample normal distribution
5:         $sleep(delay)$   ▷ thread should sleep
6:         $ActiveUserNum \leftarrow ActiveUserNum - 1$   ▷ should be locked (Mutex)
7:         return $delay\}$
8: end function
```

that can have a different type. It can be applied to construct a new generator based on the generated value of the original generator. Inside this function, the number of active users is increased to simulate a request. (The access to *ActiveUserNum* should be locked to avoid race conditions.) Then, a value is sampled according to the normal distribution and assigned to the *delay* variable. The sample is created with the parameters μ and σ from the *rtime* function that was explained before. Note that *encapsulatedAttr.length* represents the number of attributes that are set by a subtask (*#Attributes*), and $sizeOf(data)$ is the size of the generated attribute data, i.e., the *ObjSize* argument of the *rtime* function. Next, the thread is put to sleep for the duration that was generated with the *sample* function. Then, the number of users is decreased again. Finally, the generated delay is returned so that it can be checked outside the generator.

Note that this generator function also applies the generated delay. This is done, because we need to know the number of active users for the generation of a sample. In order to know which user is active, it is necessary to directly execute this behaviour, so that we have active users during the generation step. Multiple users are executed concurrently in different threads in an independent way. However, their shared variable *ActiveUserNum* causes a certain dependency between the user threads, because when one user increases this variable, then this affects the response-time distributions of the other users.

The usage profiles are also parsed and the extracted user behaviour is added into the combined model. The user input-durations that represent the time needed for filling web forms can be integrated in a similar way as the *rtime* functions by introducing input-duration generators. Their implementation details are omitted, as they work in the same way as response-time generators except that they do not change the number of active users, and they use a uniform distribution instead of a normal distribution. With both these generators, we are able to implement the sequence of subtasks of tasks as represented in Figure 6.4. Input-duration generators represent the time that a user needs for the input (e.g., for filling forms) and response-time generators simulate the response times of different requests. These generators are instantiated with different parameters depending on the request type. Algorithm 13 shows the necessary

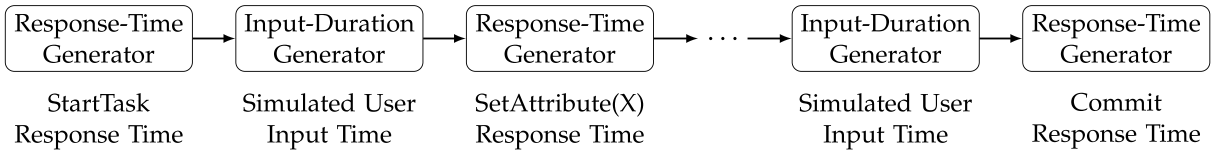


Figure 6.4: Generator sequence of a task that is executed with a sequence generator.

Algorithm 14 Pseudo code of a latency generator for MQTT.

```
Inputs: Message $msg$,
        $encapsulatedGens$,              ▷ a map ($String \to Gen$) for the data generators of a message
        $latency : (\dots) \to (\mu, \sigma)$      ▷ function for the simulation of the latency
Global Variable: $\#ActiveMsgs \in \mathbb{N}_{\geq 0}$,   ▷ number of MQTT interactions that are currently open
                 $\#TotalSubs \in \mathbb{N}_{\geq 0}$     ▷ number of total subscriptions that are managed on the broker
                 $Subs : \text{Topic} \to \mathbb{N}_{>0}$      ▷ map for the subscription numbers of a client
1: function Generator
2:     return $Gen.map(encapsulatedGens).select(data \to \{$
3:         $\#subs \leftarrow \begin{cases} Subs[data[\text{"}topic\text{"}]], \text{ if } msg = \text{"}publish\text{"} \\ 0, \text{ otherwise} \end{cases}$      ▷ only set $\#subs$ for publish
4:         $\#ActiveMsgs \leftarrow \#ActiveMsgs + 1$      ▷ should be locked (Mutex)
5:         $delay \leftarrow sample(latency(msg, \#ActiveMsgs, \#TotalSubs, \#subs)$      ▷ sample normal distr.
6:         $sleep(delay)$      ▷ thread should sleep
7:         $\#ActiveMsgs \leftarrow \#ActiveMsgs - 1$      ▷ should be locked (Mutex)
8:         return $delay\}$
9: end function
```

parameters for the instantiation. It is important to point out that the model can be simulated with a virtual time, i.e., a fraction of the actual time. Hence, the *delay* variable of Algorithm 13 should normally be divided by a constant value, but we omitted this detail for the sake of simplicity.

For MQTT, we implemented latency generators that simulate the timing behaviour of MQTT. These generators work in a similar way as response-time generators.

Algorithm 14 represents the implementation of a latency generator. The inputs are a message, a map of encapsulated generators with identifier strings as keys (e.g., for the topic and message content generation), and the *latency* function that returns the parameters μ and σ of the normal distribution and that was introduced in the previous chapter. Additionally, there are global variables *#ActiveMsgs* for the open message exchanges, *#TotalSubs* for the total number of subscriptions of the broker, and a map *Subs* for the number of subscriptions per topic. The generator is expressed as a function that is called during the generation process and it works as follows. First, $Gen.map(encapsulatedGens)$ is applied to build a generator that produces a map for the message data, which includes entries, like $\text{"}topic\text{"} \to \text{"}test\text{"}$. *Gen.map* takes a map that contains generators as input, and returns a generator that has values for these generators:

$$Gen.map : (A \to Gen[B]) \to Gen[A \to B]$$

The *select* function is applied to the resulting map generator in order to further process the generated data. Inside this function, we have access to the generated message *data* map. In case of a *publish* message, *#subs* is set to the number of subscribers for the generated topic $data[\text{"}topic\text{"}]$, or set to zero otherwise. Next, the number of active message exchanges *#ActiveMsgs* is increased. (The access to this variable should be locked to avoid race conditions.) Then, a value is sampled according to the normal distribution and assigned to the *delay* variable. The sample is created with the parameters μ and σ from the *latency* function that was explained before. Next, the thread is put to sleep for the generated latency. Finally, *#ActiveMsgs* is decreased again and the *delay* is returned so that it can be checked outside the generator.

Note that this generator also executes the generated delay (Line 6) for the same reason as for the response-time generator, i.e., we need to know the number of active messages during the generation of a sample.

The simulation of time between messages, defined in the usage profile, is much simpler than the latencies. A sample of a uniform distribution suffices to execute this delay with a

Algorithm 15 Pseudo code of the test-case generation for classical PBT and SMC.

```
Input: $spec$,                                              ▷ PBT state-machine specification
       $size \in \mathbb{N}_{\geq 0}$                       ▷ parameter for test-case length
 1: $model \leftarrow spec.initialModel()$
 2: for $i \leftarrow 1$ to $size$ do
 3:     $gen \leftarrow spec.next(model)$                   ▷ next returns a command generator
 4:     $cmd \leftarrow gen.sample()$                       ▷ command is generated
 5:     $model \leftarrow cmd.runModel(model)$              ▷ command is executed
 6: end for

 7: function spec.next(model)
 8:     $set \leftarrow model.getTransitionsWithWeights()$  ▷ set of $(weight, Gen[Transition])$
 9:     return $Gen.frequency(set).selectMany(transition \rightarrow$
10:         $Gen.map(transition.Generators).selectMany(data \rightarrow$ ▷ generate data or response times/latencies
11:             $CmdGenerator(transition, data)))$          ▷ generator for a command
12: end function
```

sleep-statement. This can be done in a dedicated generator or just in the execution functions of commands, which handle the execution of messages or requests and which were explained in Section 3.3.1.

The selection of tasks or messages according to weights of the usage profile was implemented with a standard frequency generator. It takes a set of weight-generator pairs and selects one of the generators according to the weights.

$$Gen.frequency : \mathcal{P}(\mathbb{R}_{>0} \times Gen) \rightarrow Gen$$

This generator was applied in the *next* function of the state-machine specification, which performs the selection of commands in order to produce command sequences, as explained in Section 3.3.1. The generator for commands does not only generate commands, but also their required data.

We implemented a test-case generation process that can do both, classical PBT for producing log data, as well as SMC of our timed models. For the log creation, we apply normal data generators to produce test cases that are executed on the SUT. For SMC of the timed model, we apply latency generators and analyse the produced test cases.

Algorithm 15 outlines this process. It requires a state-machine specification *spec*, which includes a generator for commands and the initial state of the model. First, the initial model is retrieved from a function of the *spec*. Then, there is an iteration over the *size* parameter and in each iteration the *next* function of the *spec* is called to obtain a command generator for the current model state. A command *cmd* is produced with this generator (Line 4) and executed on the model *cmd.runModel* in order to retrieve a new model, which incorporates the applied state change. Then, this new model (state) is given to the *next* function in each iteration in order to produce a command sequence. The next function works as follows: First, a set of pairs of weights and transition generators is retrieved from the *getTransitionsWithWeights* function. Based on this set, a frequency generator is built (Line 8).

The function *selectMany* of this generator is called to further process the selected value. This function was already explained in Section 3.5. It is applied to a generator in order to build a new generator. Within this function, a map generator is built that generates a data map for the transition. For example, the map can contain a topic for a message, when we directly test an MQTT broker, or a latency for the analysis of our timed model. The *selectMany* function is applied again on this generator and within this function, a command generator is created for the given transition and data. A resulting test case is a sequence of command and model instances, e.g., it contains all the information of a timed trace that can be analysed with SMC.

We apply the state-machine specification with the described *next* function in order to produce a state-machine property that performs the aforementioned test-case generation. Then, we execute this property within our SMC properties that perform an SMC algorithm. SMC properties were introduced for the integration of SMC into PBT as described in Chapter 4. More specifically, for prediction with our models, we apply *ChernoffProperties* that are depicted in Algorithm 10, and for testing the predictions we apply *SPRTProperties* as illustrated in Algorithm 11.

In the next chapter, we present the evaluation of our method, which is supported by our implemented generators and by the developed test-case generation algorithm.

7 Evaluation of the Response-Time Prediction and Testing Method

This chapter is based on our publications at QEST 2018 [3], at SETTA 2018 [12], and in the journal SQJO 2017 [9].

In this chapter, we present the evaluation of our response-time prediction and testing method. We illustrate the viability and generality of our method with our two case studies (TFMS and for MQTT). For the TFMS, we show how to assess the prediction power of our models by performing two hypothesis tests. Moreover, we demonstrate a further application of our method for deployment testing in order to test if system deployments with a different hardware or network setup have a performance comparable to that of a reference SUT. For MQTT, we present a performance comparison between different broker implementations. The utilisation of our method for this application area allows us to check which broker shows the better performance, depending on a specific usage scenario. In order to perform such a check, we evaluate both broker implementations with two different usage profiles.

7.1 TFMS

We evaluated our method for two major modules of the TFMS, the Test Order Manager and the Test Equipment Manager. These modules were already discussed in Section 3.6, where we also presented their underlying functional models in detail. Now, we demonstrate a performance evaluation of these modules. We focus on the response times and the number of samples needed, and also present run times of the simulation and testing process.

7.1.1 Settings.

The evaluation was performed in a distributed environment at AVL. The TFMS server (version 1.8) was running on a virtual machine with Windows Server 2012, 15 GB RAM and 7 Intel Xeon E5-2690v4 2.6 GHz CPUs. The test clients that simulated the users were executed in a separate virtual machine with Windows Server 2008, 6 GB RAM and 3 Intel Xeon E5-2690v4 2.6 GHz CPUs. The logs for the linear regression were created on these test clients, and they were applied to evaluate our models. For both, the test-case generation to create the logs and the simulation with SMC, we applied FsCheck version 2.8.2.

7.1.2 Test Order Manager

The Test Order Manager is the main module of our SUT. It enables the configuration and execution of test orders, which are basically a composition of steps that are necessary for a test sequence at an automotive test bed. The functional models of this module were already presented in Section 3.6.2 and the extension of these models to stochastic timed automata works in the same way as illustrated with the example in Section 6.1.

We applied our method in order to answer the following question: "What is the probability that the response time of all requests within a task sequence of a fixed length, i.e., a test case, is under a specific threshold for each user within a population?". For this evaluation, a usage profile was created in cooperation with domain experts from AVL. This profile was similar to the one shown in Section 4.2, and is illustrated in Listing 7.1. The multiple linear regression model was similar to the one of Section 4.2 as well and it is shown in Listing 7.2. The only major difference is that this regression model is larger due to the higher number of transitions

```
{TaskWeights:{
   TestOrder:{ToCreate:35, MakeReady:1, Finish:1, Activate:1, Duplicate:1, Reject:1,
              AdminEdit:1, EditCreated:65, EditStandardWorkInWork:65,
              EditStandardWorkExecuted:65, CancelInCreated:1,CancelInStandardWorkInWork:1,
              CancelInStandardWorkExecuted:1,CancelInFinished:1, CancelInInvalid:1,
              Invalidate:1, Select:5,SelectREM:1},
   BusinessProcessTemplate:{BptCreate:35, AdminEdit:1, Edit:65, ChangeState:1, Select:5,
                           SelectREM:5}},
TaskWaitIntervalStart:500, TaskWaitIntervalEnd:1500, SubTaskWaitIntervalStart:300,
SubTaskWaitIntervalEnd:500, WaitPerReference:10, WaitPerCharacter:30 }
```

Listing 7.1: Usage profile of the Test Order Manager.

```
                                           Estimate  Std. Error  t−value     Pr(>|t|)
(Intercept)                                 25.570      0.194     131.200     0.0
#Users                                       4.980      0.016     299.814     0.0
#Attributes                                  2.473      2.477       0.998     0.318
ObjectSize                                   0.000      0.003       0.054     0.956
CumulativeObjectSizeForRisingTasks           1.291e−06  4.972e−09 259.800     0.0
AdminEdit_SetRefAttribute                    4.890      0.154      31.592     1.302e−218
AdminEdit_StartTask                          0.728      0.639       1.138     0.254
BptCreate_Commit                            −7.968      0.248     −32.027     1.343e−224
BptCreate_SetRefAttribute                    4.353      0.165      26.349     8.694e−153
BptCreate_StartTask                          0.371      0.642       0.577     0.563
CancelInCreated_Commit                       3.474      1.455       2.387     0.016
CancelInCreated_StartTask                    1.278      0.678       1.885     0.059
CancelInFinished_Commit                      3.483      1.663       2.094     0.036
CancelInFinished_StartTask                  −4.583      1.038      −4.414     1.013e−05
CancelInStandardWorkExecuted_Commit          2.102      1.941       1.082     0.278
CancelInStandardWorkExecuted_StartTask      −3.765      1.428      −2.636     0.008
CancelInStandardWorkInWork_Commit            3.337      1.517       2.199     0.027
CancelInStandardWorkInWork_StartTask        −4.357      0.792      −5.500     3.794e−08
ChangeState_Commit                          −1.231      5.335      −0.230     0.817
ChangeState_SetRefAttribute                  4.484      0.245      18.267     1.677e−74
ChangeState_StartTask                        1.055      0.656       1.608     0.107
Duplicate_Commit                            56.725      0.243     233.013     0.0
Duplicate_SetRefAttribute                    4.779      0.227      21.032     4.099e−98
Duplicate_StartTask                          2.405      0.653       3.683     0.000
EditCreated_Commit                           2.253      0.315       7.149     8.720e−13
EditCreated_SetRefAttribute                  4.627      0.289      16.004     1.267e−57
EditCreated_StartTask                        0.855      0.680       1.256     0.208
EditStandardWorkExecuted_Commit              3.400      1.712       1.985     0.047
EditStandardWorkExecuted_StartTask          −4.032      1.123      −3.588     0.000
EditStandardWorkInWork_Commit                4.678      1.521       3.075     0.002
EditStandardWorkInWork_StartTask            −3.125      0.793      −3.937     8.232e−05
Edit_Commit                                 −2.849      0.371      −7.661     1.847e−14
Edit_SetRefAttribute                         5.042      0.346      14.572     4.389e−48
Edit_StartTask                               4.038      0.706       5.713     1.108e−08
Finish_Commit                                4.223      1.773       2.381     0.017
Finish_StartTask                            −2.666      1.208      −2.206     0.027
MakeReady_Commit                             4.294      1.451       2.958     0.003
MakeReady_SetRefAttribute                    5.717      0.335      17.050     3.794e−65
MakeReady_StartTask                          1.395      0.671       2.076     0.037
Reject_Commit                                4.792      1.515       3.162     0.001
Reject_StartTask                            −4.272      0.784      −5.442     5.264e−08
SelectREM_Load                              −9.469      0.637     −14.848     7.449e−50
SelectREM_Open                              −5.106      0.637      −8.007     1.178e−15
Select_Load                                 −9.177      0.636     −14.410     4.675e−47
Select_Open                                 −5.456      0.636      −8.567     1.064e−17
ToCreate_Commit                             −4.903      0.335     −14.617     2.272e−48
ToCreate_SetRefAttribute                     6.886      0.141      48.576     0.0
ToCreate_StartTask                          −2.340      0.714      −3.278     0.001
Attribute_NOTSET                           −15.211      0.588     −25.847     4.094e−147
Attribute_ParentFolder                     −31.078      0.220    −140.743     0.0
...
```

Listing 7.2: Linear regression model of the Test Order Manager.

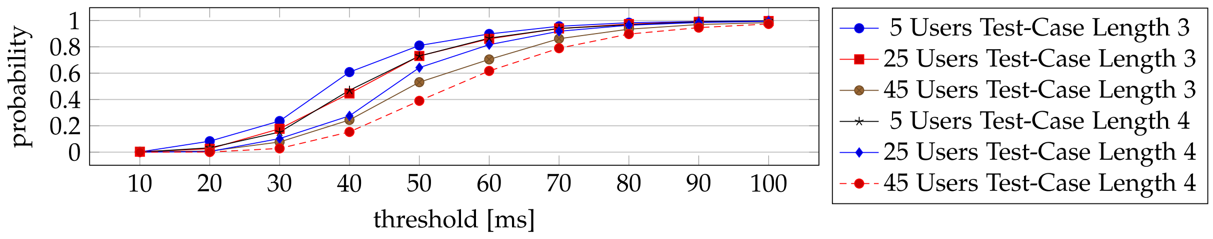


Figure 7.1: Test Order Manager simulation results of the model.

in the Test Order Manager model. Note that more accurate usage profiles could be obtained by monitoring a live system with real users. Unfortunately, this was not possible in our case, because we did not receive approval from TFMS customers.

We applied the profile to form user populations of different sizes, and we applied our evaluation for test cases with increasing lengths via a Monte Carlo simulation with Chernoff-Hoeffding bound with parameters $\epsilon = 0.05$ and $\delta = 0.01$. (This requires 1060 samples per data point.) The results for an empty database ($CumulativeObjSize = 0$) and for a database size that represents about 14,000 test orders ($CumulativeObjSize = 80,000,000$) are shown in Figure 7.1 and Figure 7.2. Note, we selected the user-population sizes $(5, 25, 45)$ by starting from a trivial size of five and by choosing a step size that showed a significant difference.

As expected, a decrease in the probability of our given question can be observed, when the test-case length or the population size increases. Moreover, it is apparent that the size of the database has an important influence on the response times. We can see that the response times increase when the database size rises. The advantage of the simulation on the model-level is that it runs much faster than on the SUT. With a virtual time of $1/10$ of the actual time, we can perform simulations that would take days on the SUT within hours.

It is also important to check the probabilities that we received through model simulation on the SUT. This was done as explained in Section 4.2 by applying the SPRT with the same parameters. Table 7.1 shows the results. Due to the high computation effort, we only check a limited selection of data points of Figure 7.1. The table shows the hypotheses and evaluation results for different thresholds, different numbers of users and for the two database fill levels ($CumulativeObjSize$). As explained in Section 6.2, we perform two SPRTs, one to check if the SUT is not much worse than the model, and one to check if the SUT is not much better than the model. The alternative hypothesis H_1 is produced via the model simulation and is the same in both SPRTs, but the null hypotheses are different (smaller: 1. H_0 or larger: 2. H_0). As result, we report the accepted hypotheses. When not all users accepted the same hypothesis, we report the respective number how often H_0 and H_1 have been accepted. Moreover, we show the average number of samples that were needed for the SPRT (#Samples) and the total run time for all clients of this evaluation. We only perform one SPRT if the predicted probability of the model is close to one or zero, because then we are already close enough to the minimum or maximum probability.

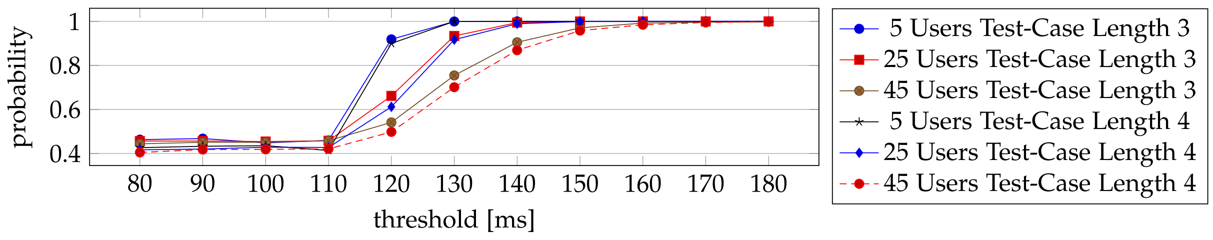


Figure 7.2: Test Order Manager simulation results of the model with filled DB.

Table 7.1: Test Order Manager results of the SUT evaluation with the SPRT.

Threshold [ms]	#Users	Cumulative ObjSize	H_1	1. H_0	Result	#Samples	2. H_0	Result	#Samples	Time [min:s]
50	5	0	0.728	0.478	H_1	20.6	0.978	$1H_1\,4H_0$	14.6	17:56
50	25	0	0.653	0.403	H_1	11.76	0.902	H_0	29.48	51:30
50	45	0	0.456	0.206	H_1	7.68	0.705	H_0	17	22:02
100	5	0	0.997	0.747	H_1	16	-	-	-	11:22
100	25	0	0.995	0.745	H_1	16	-	-	-	11:56
100	45	0	0.986	0.736	H_1	17.84	-	-	-	22:08
100	5	80,000,000	0.428	0.178	H_1	27	0.678	H_1	37.6	28:46
100	25	80,000,000	0.425	0.175	H_1	35.56	0.675	H_1	29.16	38:54
100	45	80,000,000	0.419	0.169	$7H_0\,38H_1$	62.82	0.669	H_1	18.22	55:37
150	5	80,000,000	1	0.750	$2H_0\,3H_1$	12.8	-	-	-	10:25
150	25	80,000,000	0.999	0.749	$16H_0\,9H_1$	9.64	-	-	-	22:40
150	45	80,000,000	0.972	0.722	H_0	5.78	-	-	-	6:52
200	5	80,000,000	1	0.750	H_1	16	-	-	-	11:32
200	25	80,000,000	1	0.750	H_1	12.72	-	-	-	11:56
200	45	80,000,000	1	0.750	H_1	13.82	-	-	-	32:46

We can see that the alternative hypotheses were accepted in many cases, which means that the predicted probability was close enough to the real probability of the SUT. In some cases, H_0 was accepted, which means that our model was too optimistic or pessimistic in these cases. We will discuss this later in Section 7.4. Moreover, it is apparent that in contrast to the execution on the model, fewer samples are needed, since the SPRT stops when it has sufficient evidence. The smaller number of required samples of the SPRT (max. appr. 62) compared to Monte Carlo simulation (1060 samples) allowed us to analyse the SUT within a feasibly short time. For example, in the worst case it took only about an hour to apply the SPRT.

7.1.3 Test Equipment Manager

The Test Equipment Manager is another important module of our SUT. This module enables the administration of equipment that is relevant for the test beds, like measurement devices, sensors, actuators and various input/output modules. The functional models and a detailed description of this module were already presented in Section 3.6.3.

We performed the same evaluation for the Test Equipment Manager as for the Test Order Manager. The usage profile (Listing 7.3) and the regression model (Listing 7.4) were also similar to the one shown in Section 4.2. The results of the Monte Carlo simulation for an empty database ($\mathit{CumulativeObjSize} = 0$) and for a database size that represents about 9,200 test equipment objects ($\mathit{CumulativeObjSize} = 30,000,000$) are presented in Figure 7.3 and Figure 7.4. We can see that the curves for an empty database are similar to that of the Test Order Manager. The curves for a filled database are different, i.e., we can see that there is a larger gap between the curves for specific numbers of users and that the response times with the filled database are much higher than those with the empty database. This difference is caused by a higher number of subtasks that are dependent on the database size in this module.

We also evaluated the results of the Monte Carlo simulation in the same way as before by applying the SPRT. Table 7.2 shows the results. For the empty database, we see that H_1 was accepted in most of the cases, but for the filled database, H_0 was accepted more often. The model seems to be too optimistic for this database size. We think the reason for this is that we have much more subtasks that are dependent on the database size. In addition, more data is transferred over the network in comparison with the Test Order Manager. This causes more network interference and makes the linear regression more difficult. Nevertheless, it was again possible to evaluate the SUT by applying the SPRT with an acceptable number of samples (max. ca. 20) and with a decent run time (max. ca. 13 minutes).

```
{TaskWeights:{
   TestEquipmentType:{TetCreate:20, TetEditGeneral:80, TetAdminEdit:1, TetChangeState:1,
                      Select:5, SelectREM:1},
   TestEquipment:{CreateTestEquipment:20, TeEditGeneral:80, MarkAsDefect:1, Select:5,
                  SelectREM:5}},
TaskWaitIntervalStart:500, TaskWaitIntervalEnd:1500, SubTaskWaitIntervalStart:300,
SubTaskWaitIntervalEnd:500, WaitPerReference:10, WaitPerCharacter:30 }
```

Listing 7.3: Usage profile of the Test Equipment Manager.

```
                                        Estimate    Std. Error   t value     Pr(>|t|)
(Intercept)                             24.170      0.735        32.873      8.301e-236
#Users                                  12.583      0.079        158.534     0.0
#Attributes                             -18.201     11.522       -1.579      0.114
ObjectSize                              0.007       0.014        0.507       0.611
CumulativeObjectSizeForRisingTasks      2.962e-06   1.604e-08    184.665     0.0
CreateTestEquipment_SetRefAttribute     33.426      0.789        42.360      0.0
CreateTestEquipment_StartTask           1.713       2.550        0.671       0.501
MarkAsDefect_Commit                     12.167      6.812        1.786       0.074
MarkAsDefect_StartTask                  12.630      2.618        4.823       1.414e-06
SelectREM_Load                          -1.909      2.539        -0.751      0.452
SelectREM_Open                          -4.122      2.539        -1.623      0.104
Select_Load                             -1.096      2.538        -0.432      0.665
Select_Open                             -1.701      2.538        -0.670      0.502
TeEditGeneral_Commit                    -2.073      14.436       -0.143      0.885
TeEditGeneral_StartTask                 11.302      2.641        4.279       1.876e-05
TetAdminEdit_Commit                     -21.109     24.536       -0.860      0.389
TetAdminEdit_SetRefAttribute            7.646       0.500        15.279      1.187e-52
TetAdminEdit_StartTask                  8.616       2.582        3.336       0.000
TetChangeState_Commit                   -36.918     6.842        -5.395      6.837e-08
TetChangeState_StartTask                6.045       2.627        2.300       0.021
TetCreate_Commit                        -12.211     10.486       -1.164      0.244
TetCreate_SetRefAttribute               12.538      0.450        27.808      1.416e-169
TetCreate_StartTask                     0.856       2.553        0.335       0.737
TetEditGeneral_Commit                   -46.019     6.847        -6.720      1.817e-11
TetEditGeneral_StartTask                9.978       2.622        3.805       0.000
Attribute_NOTSET                        -29.440     1.840        -15.993     1.653e-57
Attribute_ParentType                    20.184      0.530        38.078      0.0
...
```

Listing 7.4: Linear regression model of the Test Equipment Manager.

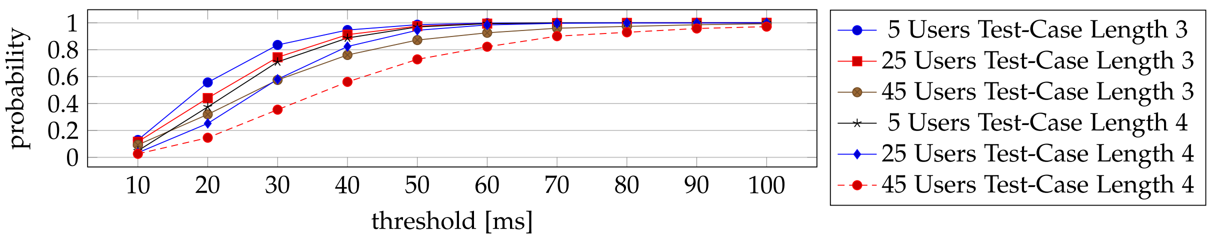


Figure 7.3: Test Equipment Manager simulation results of the model.

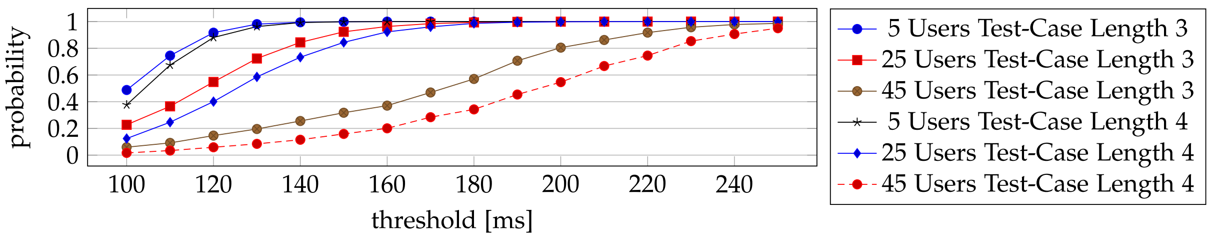


Figure 7.4: Test Equipment Manager simulation results of the model with filled DB.

Table 7.2: Test Equipment Manager results of the SUT evaluation with the SPRT.

Thresh-old [ms]	#Users	Cumulative ObjSize	H_1	1. H_0	Result	#Sam-ples	2. H_0	Result	#Sam-ples	Time [min:s]
50	5	0	0.973	0.723	H_1	19.6	-	-	-	10:52
50	25	0	0.936	0.686	H_1	16.2	-	-	-	9:46
50	45	0	0.671	0.421	H_1	11.11	0.921	H_0	18.58	13:11
100	5	0	1	0.750	H_1	16	-	-	-	5:46
100	25	0	0.998	0.748	H_1	16	-	-	-	6:10
100	45	0	0.962	0.712	H_1	16	-	-	-	7:49
100	5	30,000,000	0.013	0.125	H_0	14	0.625	H_1	9	3:39
100	25	30,000,000	0.114	-	-	-	0.364	H_1	14.4	2:59
100	45	30,000,000	0.014	-	-	-	0.264	H_1	16.24	3:11
150	5	30,000,000	0.999	0.749	H_0	5	-	-	-	3:19
150	25	30,000,000	0.820	0.570	H_0	6	1	H_1	5	3:55
150	45	30,000,000	0.137	0.387	H_0	14.27	-	-	-	3:15
200	5	30,000,000	1	0.750	$3H_0\,2H_1$	12.8	-	-	-	4:41
200	25	30,000,000	0.997	0.747	H_0	5.68	-	-	-	4:58
200	45	30,000,000	0.496	0.246	H_0	13.56	0.746	H_1	7.69	6:00

7.1.4 Run Times of the Method

Our method consists of several phases that have different computation times. Here, we give an overview of the timings of these phases in order to illustrate the overall run time of our method and to demonstrate its effectiveness.

In the first step, we generate log data with MBT. This initial testing phase took about an hour for both our tested modules, i.e., about 63 minutes for the Test Order Manager and about 65 minutes for the Test Equipment Manager. The next step was the linear regression, which took only about 70 to 100 seconds including the time for data cleaning and preprocessing.

The model-simulation times are illustrated in Table 7.3. Note that these timings were measured on the client machine that was described in the setting. It can be seen that they were very similar for the empty and the filled database. The reason for this is that the user-input times from the usage profiles accounted for the bulk of the simulation time. For the same reason, we only see a small increase in the simulation time, when the number of users rises. In summary, the simulation time was about 9 to 13 minutes for the Test Order Manager and 6 to 9 minutes for the Test Equipment Manager.

The last columns of Table 7.1 and Table 7.2 show the run times of the SPRTs. Note that during the execution of a sample, we stopped when we already observed a higher response time than our threshold, and we only have one run time for both SPRTs, since we check them in one execution. The run times of the Test Order Manager were about one hour in two cases. In all other cases, run times were mostly shorter than half an hour and the fastest experiments

Table 7.3: Average simulation time [min:s] of the model for the Test Order Manager and the Test Equipment Manager for an empty and filled database.

#Users	Test-Case Length	Simul. Time (Empty DB)		Simul. Time (Filled DB)	
		Test Order Manager	Test Equipment Manager	Test Order Manager	Test Equipment Manager
5	3	9:24	6:40	9:23	6:40
25	3	9:31	6:51	9:41	6:51
45	3	9:37	7:08	9:45	7:08
5	4	12:46	8:56	12:45	8:57
25	4	12:52	9:09	12:58	9:09
45	4	13:02	9:37	13:04	9:37

Table 7.4: Different system deployments with various hardware/network settings.

Deployment	Hardware		Network	
	#CPUs	RAM [GB]	Bandwidth [Mbps]	Delay [ms]
D_0	7	15	1000	0
D_1	7	4	1000	0
D_2	2	15	1000	0
D_3	7	15	500	0
D_4	7	15	100	0
D_5	7	15	50	0
D_6	7	15	1000	25
D_7	7	15	1000	10

took about 10 minutes The run times of the Test Equipment Manager were shorter due to its lower complexity, i.e., a smaller number attributes for the form data. They were always below 15 minutes and in the fastest cases about 3 minutes.

Executing the Monte Carlo simulation that we applied for the model directly on the SUT would take about one day. By applying the SPRT, we can perform such an evaluation within less than an hour in the worst case.

7.2 Deployment Testing

A further application area of our method is deployment testing. For this technique, we apply the same steps as for the evaluation of the previous section, which are described in Chapter 5 and Chapter 6. The only difference is that we perform the hypothesis tests on deployments with different hardware settings and that we generate the log data for the linear regression on a reference system.

Our aim is to evaluate the performance on this reference system and then based on this evaluation, we check if deployments with different hardware or network settings have a comparable performance. Therefore, we simulate the model of the reference system in order to derive hypotheses about the expected performance that can then be tested on the deployments. For these tests, the sequential probability ratio test is performed as described in Section 2.2.3, because it enables an efficient evaluation with a small number of samples.

In order to evaluate this approach, we applied it to the Test Order Manager similarly as explained in Section 7.1.2.

Test Setup. We evaluated a TFMS server (version 1.8) that was running on a virtual machine with Windows Server 2012. Our reference SUT (D_0) had 15 GB of RAM and 7 Intel Xeon E5-2690v4 2.6 GHz CPUs. A similar virtual machine with 6 GB RAM and 3 CPUs was used to run the test clients. We defined a set of deployments by varying values for the CPUs, the RAM size, the network bandwidth, and the network delay. These deployments (D_i) are shown in Table 7.4. Since the server was running on a virtual machine, the hardware settings could easily be changed. A tool called Network Emulator for Windows helped us to configure the network setup of the test client, e.g., it allowed us to decrease the network bandwidth.

Monte Carlo Simulation of the Model. We applied our method in the same way as for the Test Order Manager case study of the previous section, i.e., to answer the question: "What is the probability that the response time of all requests within a task sequence of a fixed length (a test case) is under a specific threshold for each user within a population?". For this analysis, we used the same usage profile (Listing 7.1) and also the same model as in Section 7.1.

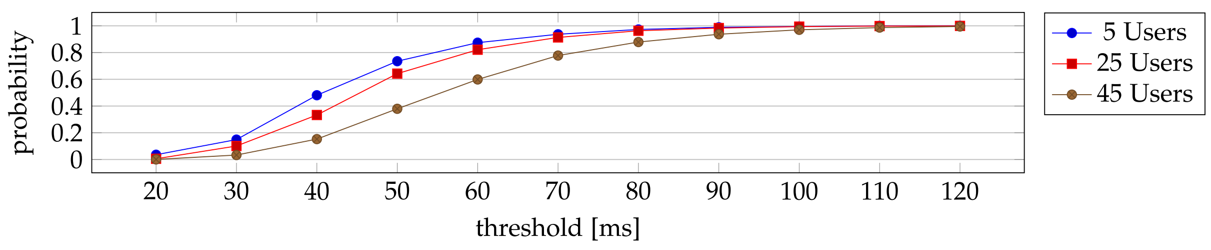


Figure 7.5: Test Order Manager Monte Carlo simulation results of the model.

We evaluated user populations of different sizes and checked a number of response-time thresholds with a fixed test-case size of four tasks. The evaluation was performed with a Monte Carlo simulation with Chernoff-Hoeffding with 1060 samples, but this time we are only interested in a smaller set of data points than in the previous section. Figure 7.5 shows the results. Same as in the last section, a decrease in the probability of our given question can be observed, when the number of users increases or the threshold decreases.

Hypothesis Testing with the SPRT. Next, we used the probabilities of the Monte Carlo simulation as hypotheses (H_1) for SPRTs of the different deployments. We selected six data points of Figure 7.5 with interesting thresholds and different user numbers in order to form the hypotheses shown in Table 7.5. We evaluated all deployments as explained in Section 6.2 by applying the SPRT with the same parameters. Figure 7.6 summarises the results in three groups: one for the deployments (and SPRTs), where all clients accepted H_1, one where there was no clear consensus among the clients, and one where all clients accepted H_0. It can be seen that H_1 was accepted by most of the deployments, which means that they provide a similar performance. For one deployment (D_5), only SPRT 1–4 were successful, SPRT 5–6 were inconclusive, i.e., 48 % of the clients accepted H_1 for SPRT 5 and 44 % for SPRT 6. For two deployments, H_0 was accepted, which means that their response times were worse than that of the reference SUT. In summary, it can be said that a change in the server hardware did not adversely affect the performance, as H_1 was accepted for all deployments with a changed hardware. Also, a change in the network bandwidth had only a weak influence on the performance. A clear change in the performance was only observed for deployments with a higher network delay.

Table 7.5: Different SPRTs for various numbers of users and thresholds.

SPRT No.	#Users	Threshold [ms]	H_0	H_1
1	5	50	0.478	0.729
2	25	50	0.400	0.650
3	45	50	0.201	0.451
4	5	100	0.746	0.996
5	25	100	0.744	0.994
6	45	100	0.738	0.988

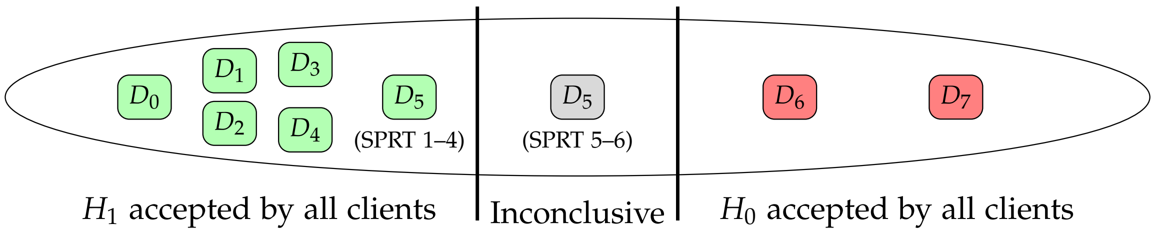


Figure 7.6: SPRT results of the different deployments.

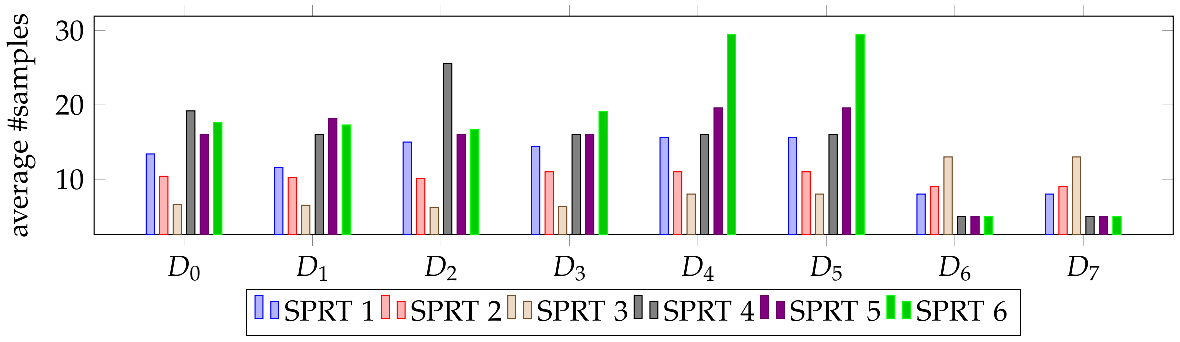


Figure 7.7: Average number of samples (test cases) for the SPRTs of our deployments.

Additionally, we evaluated the number of needed samples from the SPRTs. Note that in order to obtain an average number of needed samples, we run the SPRT concurrently for each user of the population and calculate the average of these runs. Multiple independent SPRT runs would produce a better average, but the computation time was too high. Figure 7.7 shows the average number of needed samples for the SPRTs of different deployments. It can be seen that certain SPRTs are quite easy to check, e.g., SPRT 3 only needs about 6–13 samples, other SPRTs take more than twice as many samples. However, a maximum of about 30 samples is still very low compared to the 1060 samples of the Monte Carlo simulation. This low number of samples allows us to evaluate multiple SUT deployments within a feasible time.

7.3 MQTT

We performed another evaluation of our method for testing protocols of the Internet of Things. More precisely, we tested two open-source MQTT implementations: Mosquitto and emqtt.

7.3.1 Settings

The evaluation was performed with Mosquitto version 1.4.15 and emqtt version 2.3.5, running with quality of service level one and in their default configurations. We analyse the needed number of samples and the run times. MQTT implementations typically have various settings, e.g., the length of the in-flight message queue or an option to group together TCP packets (Nagle's algorithm [135]). The influence of such settings might be a potential threat to the validity of our comparison. We worked with the default settings as this is commonly done, and we also tried to adapt the mentioned settings to face this threat. A comparison of the regression models and response-time visualisations did not show a difference for the adapted settings. Note that Nagle's algorithm has no effect, because it only groups messages if acknowledgements are pending. This situation does not occur, since our tests are synchronous, i.e., we always wait for an acknowledgement before sending a new message.

The evaluation was performed on a Windows server (version 2008 R2) with a 2.1 GHz Intel Xeon E5-2620 v4 CPU with 8 Cores and 32 GB RAM. This machine was running the clients and the broker in order to avoid an influence of the network. However, a possible influence of the client processes on the broker might cause a threat to the validity of our evaluation. To face this issue, we measured the CPU load, to make sure that it is not a bottleneck. During the evaluation, the CPU load was below 60% most of the time, and there were only some rare peaks, where the CPU load was over 90%. We also tried to increase the priority of the broker process, but this showed no difference. The RAM usage of the brokers was insignificant since the total RAM of the servers was more than enough.

```
                      Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)         −4.7215355  0.0746994  −63.207   <2e−16 ***
Msgdisconnect        4.8297199  0.0830489   58.155   <2e−16 ***
Msgpublish           5.6995651  0.0941305   60.550   <2e−16 ***
Msgsubscribe         5.7603881  0.0957232   60.178   <2e−16 ***
Msgunsubscribe       5.3875122  0.0870509   61.889   <2e−16 ***
#ActiveMsgs          1.2086416  0.0032501  371.880   <2e−16 ***
#TotalSubs          −0.0001275  0.0001398   −0.912    0.362
#Subs                0.1726546  0.0197298    8.751   <2e−16 ***
```

Listing 7.5: Linear regression output (excerpt) for the MQTT broker emqtt.

We applied Visual Studio 2012 with .NET framework 4.5, NUnit 2.64, and FsCheck 2.92 in order to run the tests and for SMC. The library M2Mqtt[17] served as a client interface to facilitate the interaction with the brokers.

7.3.2 Results

We follow the method of Section 4.2 in order to answer the question "What is the probability that the message latency is under a certain threshold?". Hence, we check the probability that all messages within a sequence of ten messages for all clients of an MQTT setup have a latency under this threshold. We perform the analysis as shown in Section 4.2, with the difference that we test Mosquitto and emqtt, and we check various thresholds and different numbers of clients. We apply the same usage profile as before and the regression model for emqtt was similar to the one shown for Mosquitto that was presented in Section 5.2.4. It is displayed in Listing 7.5 and the only major difference to Mosquitto was that the *#TotalSubs* feature showed no significance as indicated by the missing *** at the end of the line. Therefore, it could be omitted as explained before. Additionally, we evaluated another usage profile (UP2), as shown in Listing 7.6, that has a higher weight for *publish* messages and different bounds for the time between messages.

As shown before, we apply a Monte Carlo simulation with 1060 samples to evaluate the timed model. The results for Mosquitto and emqtt for both usage profiles are shown in Figure 7.8 and Figure 7.9. Table 7.6 shows the average time needed for these evaluations.

As expected, a decrease in the probability can be observed, when the number of clients increases, and a higher threshold causes a higher probability. It can be seen that the curves in the figures for both usage profiles (and both brokers) are similar. The reason is that for both usage profiles costly message types, like connect or publish, are selected frequently and have a similar impact on the resulting probability in both cases.

The advantage of applying SMC on a model is that it runs much faster than on the SUT. With a virtual time of 1/10 of the actual time, we can perform evaluations that would take hours on the SUT within minutes.

It is also important to check the probabilities that we received through SMC of the timed model on the SUT. This was done as explained in Section 4.2 with hypothesis testing with the

[17] `https://m2mqtt.wordpress.com` (visited on 2018-09-19)

```
MinTimeBetwMsg: 50, MaxTimeBetwMsg: 250,
MsgWeights:{connect: 1, disconnect: 1, publish: 7, subscribe: 1, unsubscribe: 1}
```

Listing 7.6: MQTT usage profile UP2 with more frequent publish messages.

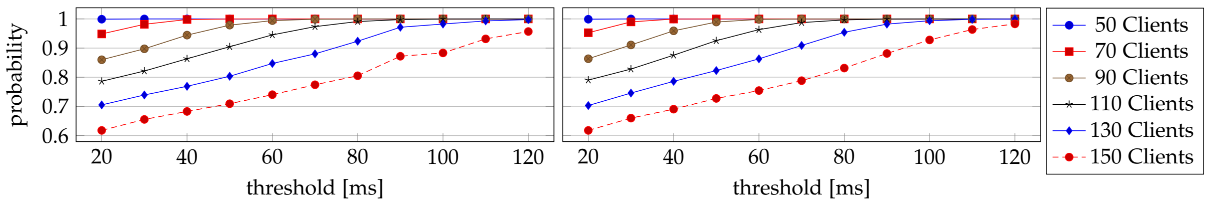


Figure 7.8: UP1 Monte Carlo simulation results for Mosquitto (left) and emqtt (right).

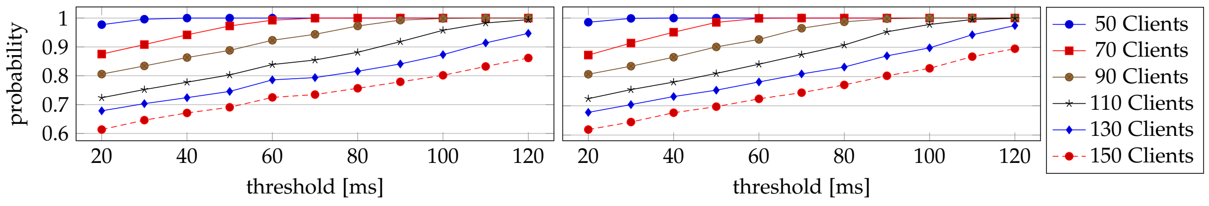


Figure 7.9: UP2 Monte Carlo simulation results for Mosquitto (left) and emqtt (right).

SPRT. Table 7.7 and Table 7.8 show the results for both usage profiles and brokers. We focused on some of the more interesting data points for the evaluation. The tables show hypotheses, test results, the needed number of samples and execution times for different numbers of clients and thresholds. Note that, in order to obtain an average number of needed samples, we run the SPRT concurrently for each client and calculate the average of these runs.

In most cases, hypothesis H_1 was accepted for almost all clients, which means that the probability of the SUT was at least as high, as the predicted one from the model. However, the prediction was not always accurate. H_0 was also sometimes accepted and in some cases H_1 was only accepted by a fraction of the clients that tested this hypothesis, e.g., for Mosquitto with a threshold of 30 ms and 90 clients, only 60% of the clients accepted H_1 for UP1. The prediction was sometimes inaccurate for small latency thresholds. The reason might be that we mainly learned the latency distributions under conditions with high load, and hence, our model might not be completely accurate for small latencies. Moreover, the prediction performed rather poorly for high numbers of clients (≥ 130), especially for UP2. This might be caused by the fact that the initial testing phase for log data had only a maximum of 100 clients and the higher number of clients might be too different from this initial test phase. However, H_1 was still accepted for most data points, which means that the model was good enough in these cases. Furthermore, it is apparent that the SPRT can be performed with fewer samples, i.e., we need mostly about 50 samples (except for some outliers), compared to the 1060 for the Monte Carlo simulation.

By comparing the results of Mosquitto and emqtt, it can be seen that predicted probabilities are too similar to make a clear distinction. However, the evaluation of the SUT with hypothesis testing was able to find some differences, i.e., in some cases emqtt showed a slightly better performance. For example, the second data row of Table 7.7 shows that Mosquitto was not able to accept H_1, where emqtt accepted it, although the same hypotheses were tested. This

Table 7.6: Average time [min:s] for the Monte Carlo simulation of the model.

#Clients	UP1 Mosquitto	UP1 emqtt	UP2 Mosquitto	UP2 emqtt
50	4:27	4:28	2:39	2:39
70	4:48	4:49	3:03	3:02
90	4:54	4:57	3:16	3:18
110	5:00	5:05	3:22	3:27
130	5:09	5:15	3:40	3:41
150	5:25	5:23	3:51	3:55

Table 7.7: Results of the evaluation of the SUT with the SPRT for usage profile UP1.

Thresh-old [ms]	#Clients	Mosquitto					emqtt				
		H_0	H_1	Result	#Sam-ples	Time [min:s]	H_0	H_1	Result	#Sam-ples	Time [min:s]
30	50	0.90	1	H_1	44	2:31	0.90	1	H_1	44	2:28
30	70	0.88	0.98	H_0	22.47	5:43	0.88	0.98	H_1	44.14	2:51
30	90	0.79	0.89	60% H_1	276.31	39:12	0.80	0.90	H_1	41.02	2:56
30	110	0.74	0.84	H_1	73.26	7:22	0.72	0.82	H_1	42.55	3:40
30	130	0.68	0.78	H_0	46.68	11:33	0.64	0.74	H_1	77.92	9:21
50	50	0.90	1	H_1	44	2:10	0.90	1	H_1	44	2:06
50	70	0.90	1	73% H_1	43.53	10:01	0.90	1	H_1	44	2:09
50	90	0.88	0.98	H_1	50.47	4:18	0.88	0.98	H_1	43	2:30
50	110	0.80	0.90	H_1	41.35	3:19	0.84	0.94	H_1	41.25	2:50
50	130	0.74	0.84	H_1	41.15	3:12	0.75	0.85	H_1	38.41	2:37
70	50	0.90	1	H_1	44	2:04	0.90	1	H_1	44	2:33
70	70	0.90	1	H_1	44	2:10	0.90	1	H_1	44	2:08
70	90	0.90	1	H_1	44	2:37	0.90	1	H_1	44	2:29
70	110	0.88	0.98	H_1	43.16	2:57	0.89	0.99	H_1	44.38	3:14
70	130	0.78	0.88	H_1	39.32	3:00	0.83	0.93	H_1	41.21	2:37

means that emqtt had a better performance in this case. For UP1, this was the case especially for small thresholds, for UP2 the performance was more similar for both implementations and there is also a case where Mosquitto showed better performance. (Row 13 of Table 7.8, shows that only 90% of the clients accepted H_1 for emqtt, but all clients for Mosquitto.)

7.3.3 Run Times of the Method

We analysed the execution times of the different phases of our method. The initial testing phase took about 5–8 minutes and the linear regression about 10–12 seconds. Note that these two phases have to be performed only once, and the resulting model can then be applied for various evaluations.

A Monte Carlo simulation of the model required about 3–5 minutes for 1060 samples as shown in Table 7.6. The evaluation of the SUT with hypothesis testing took 2–4 minutes in

Table 7.8: Results of the evaluation of the SUT with the SPRT for usage profile UP2.

Thresh-old [ms]	#Clients	Mosquitto					emqtt				
		H_0	H_1	Result	#Sam-ples	Time [min:s]	H_0	H_1	Result	#Sam-ples	Time [min:s]
30	50	0.90	1	96% H_1	42.88	1:18	0.90	1	96% H_1	43.42	1:16
30	70	0.88	0.98	H_0	17.60	1:15	0.88	0.98	H_1	46.40	2:07
30	90	0.80	0.90	H_0	17.18	3:55	0.80	0.90	H_1	44.98	1:42
30	110	0.72	0.82	H_0	13.43	1:39	0.72	0.82	H_0	14.61	1:14
30	130	0.65	0.75	H_0	14.55	0:56	0.70	0.80	H_0	12.68	0:24
50	50	0.90	1	H_1	44	1:17	0.90	1	H_1	44	1:19
50	70	0.90	1	67% H_1	37.94	3:48	0.90	1	H_1	44	1:44
50	90	0.88	0.98	H_1	51.36	3:01	0.89	0.99	H_1	46.20	1:54
50	110	0.79	0.89	H_1	41.42	1:46	0.81	0.91	87% H_1	152.34	8:34
50	130	0.72	0.82	H_0	16.46	0:58	0.76	0.86	H_0	9.51	0:23
70	50	0.90	1	H_1	44	2:07	0.90	1	H_1	44	1:16
70	70	0.90	1	H_1	44	2:11	0.90	1	H_1	44	1:18
70	90	0.90	1	H_1	46.04	2:41	0.90	1	90% H_1	52.81	2:47
70	110	0.87	0.97	H_1	65.55	4:39	0.88	0.98	H_1	43.82	1:58
70	130	0.79	0.89	H_0	48.18	5:28	0.81	0.91	H_0	9.58	0:34

most cases, in some cases about 10 minutes and only in one case 39 minutes. Hence, most of our predictions could be tested efficiently in about the same time that was needed to make the prediction with the timed model.

Running a Monte Carlo simulation with 1060 samples directly on the SUT would take approximately 2–3 hours. Performing this simulation becomes quickly impractical when various data points should be analysed. Therefore, it is reasonable to use our model-based approach, because it can be executed faster.

7.4 Discussion

The evaluation showed that our simulation approach allows us to estimate the probability that a user or client can perform a sequence of system interactions without having to wait longer than a specific threshold for a response. Moreover, we demonstrated that we can check if the estimated probability is close to the real probability of the SUT with an acceptable number of samples. In some cases, however, the models were not able to estimate the probability accurately enough: the estimates were either too optimistic or too pessimistic. This indicates that the prediction errors, i.e., the deviation of the predicted from the actual response times, might be too large. Such prediction errors might have an obvious explanation, if the R^2-score was too small. However, they might also emerge, when the R^2-score appears to be high enough. These occurrences with high prediction errors and a reasonable R^2-score are hard to resolve, especially when a distributed experimental setting is considered. They might be caused by several sources:

1. *Measurement Errors*. There might be interference or noise during the test-data generation that could artificially and unintentionally increase the response times of our log data. For example, this may be caused by memory or cache misses, by varying network delays or interruptions, by blocking effects of our SUT or by operating-system influences, like scheduling.
   With such data, it might be possible to obtain an R^2-score that seems to be acceptable, if coincidently there are linear dependencies. However, our simulations with SMC would not be as accurate as the R^2-score might suggest, because these influences might not always occur in a uniform way. Such measurement errors are lower in a non-distributed environment where there are no network influences. We observed that we obtain a better model in such environments.
   For the TFMS evaluation, we had to work with the less favourable case of the distributed environment, because this was a more interesting setup for our industrial partner AVL. The MQTT evaluation was performed within a local environment, but there were other potential interference factors caused by the high number of clients.
2. *Sampling Bias*. The data generation might not be random enough, i.e., it might be unintentionally set up in a way, where relevant scenarios for the prediction were not tested frequently enough. Hence, the log data would not contain equally many entries for these scenarios, which might cause a regression model that only performs well for dominant data examples, but not for examples that are insufficiently represented in the data.
   Moreover, the test-data generation might be biased in a way, where false dependencies were introduced that do not occur in general. In this case, we might obtain an artificial correlation between variables (or features), e.g., when the number of users would be monotonically increasing over the course of an experiment, then this would cause a deceiving correlation with the database size. A random selection of the number of users would help to avoid such an issue. Additionally, it is advised to thoroughly inspect the log data with visualisations, like scatter plots, and histograms. This can help to reduce the risk of such a bias, but generally it is not possible to completely eliminate all sources for this bias [53].

An interesting observation, which might be seen also as a weakness of our approach, is that SMC seems to be inefficient when the given threshold of the response-time property to be tested is far below or far above the actual response time. In these cases, the probability of the response-time property does not vary in a significant way with the user population size. SMC wastefully computes the probability for various user population sizes, even if a single run with a fixed user population size, say one user, would be sufficient to get a similar result. This phenomenon can be clearly observed in Figure 7.2, where the probability curves of different user population sizes are very close to each other for low and high thresholds, whereas they only go apart for thresholds close to the actual response times where the user population size seems to make a difference.

Finally, it is worth mentioning that other non-linear learning approaches might also be able to further improve the prediction power of our model. Especially, if our method is applied for other application areas, then it might make sense to investigate other learning methods when the accuracy of the obtained model is not high enough. Hence, this might be a potential topic for future work.

8 Related Work

This chapter partially contains contents from our previous publications that were discussed before in Section 1.9 [3, 5, 6, 9, 12, 162].

In this section, we present related work for the major contributions of this thesis, which were presented in the previous chapters. First, we illustrate related work of our PBT approach with business-rule models, and also for MBT and PBT in general. Then, we focus on our combination method for SMC and PBT and also SMC in general. Finally, we present related approaches for our model-based performance prediction and verification method, and we also discuss performance testing and performance engineering.

8.1 Model-Based Testing of Business-Rule Models within a Property-Based Testing Tool

Model-Based Testing. MBT is a popular testing technique. Various surveys and overviews were conducted on this topic [34, 40, 58, 84, 159, 179, 180]. MBT approaches have in common that they rely on a model of a system, which is applied to generate test cases that are executed on the system to find bugs. There is an abundance of different modelling formalisms, like state chart diagrams, sequence diagrams, or process algebras. A survey of MBT that includes different modelling (or specification) formalisms was presented by Dias Neto et al. [58]. Moreover, van Lamsweerde [110] characterised different formal specification paradigms and compared their strengths and weaknesses.

All these modelling formalisms are applied to define an abstract model of an SUT. Such a model serves as a source for the test-case generation in order to produce tests that can evaluate the behaviour of the SUT. There are also numerous test-case generation techniques, like random generation, search-based techniques, generation based on coverage criteria, mutation-based techniques, etc. Utting et al. [180] gave an extensive overview of these techniques.

The test-case generation, the test-case execution, and also sometimes the model definition, are supported by tools. Various tools have been presented for different types of models and for numerous test-case generation methods. Saifan and Dingel [159] gave an overview of MBT tools and also the taxonomy of Utting et al. [180] described a number of tools. Several well-known tools are the following.

TGV [93] is a tool that takes an extended form of labelled transition systems as model, and generates test cases that can be applied for conformance testing. A similar tool that performs online conformance testing is TorX [178] (or the newer version JTorX [32]). Also Microsoft Spec Explorer [181] performs conformance testing, but with Spec# model programs.

The MBT tool Conformiq [91] takes UML state charts as input and applies symbolic execution to generate test cases that fulfil certain coverage criteria. A random search-based approach for the test-case generation is applied by Modbat [17], which works with extended finite state machines.

The UPPAAL tool family can generate test cases for timed automata models. This family contains three tools that have a focus on MBT. UPPAAL Tron [111] generates test cases randomly for online testing, where test cases are directly executed while they are generated. UPPAAL Cover [81] applies coverage-based test-case generation and UPPAAL Yggdrasil [98] creates tests offline and allows the annotation of additional scripts to the model, which can then be included in the generated test cases.

Another tool family called MoMuT (model-based mutation testing) [102] applies a test-case generation technique that is mutation-based, and it supports different kinds of input models, like timed automata (MoMuT::TA [13]), assume-guarantee contracts (MoMuT::REQS [11]), object-oriented action systems, and UML state charts (MoMuT::UML [102]).

The tool called sal-atg [75] uses models that are defined with guarded commands, and it applies symbolic model checking for the test-case generation. MaTeLo [61] applies Markov chain usage models for statistical usage testing and it has a random generation technique. A similar statistical testing tool is JUMBL [150], which also works with Markov chains.

A graphical testing tool called TPT (Time Partition Testing) [39], allows graphical modelling of hybrid systems and has graphical support for the test-case design.

AETG [50] is a testing tool that generates test data based on combinatorial testing, i.e., it generates various combinations of input parameters in order to test different scenarios.

The closest related work in the area of MBT are approaches that also apply existing system artefacts as test models or for the test-data generation. For example, some testing methods work with web-service descriptions in order to produce requests and associated test-data. Bai et al. [24], Sneed and Huang [170], and Bartolini et al. [29] presented testing approaches that take the web-service description language (WSDL) as a source for the test-case generation. In contrast to our work, WSDL only includes limited information about the system, e.g. no system states are included. Our business-rule models also include the resulting system (or object) states after a task is performed. Hence, we can also check if the resulting states are correct during the test-case execution. This enables a better inspection of the functionality of the SUT.

Several approaches have illustrated that the XML-based business process execution language (BPEL) [124, 175] or BPEL for web services (WS-BPEL) [117] can be applied as a source for testing. However, in contrast to our work, the focus of these approaches is on testing the composition of services. With our business-rule models, we can perform a more fine-grained evaluation, which includes a check of the states of individual domain objects and an assessment of the form data that is stored in the database.

A similar testing approach to our business-rule testing method was presented by Wetherall and Woodhead [191]. The approach applies an extended version of the rule markup language RuleML [186] in order to test schedules of a scheduling application against the business rules. In contrast to our method, this approach is only intended for a limited application domain, i.e., for scheduling systems. Moreover, they do not apply their rules for generating test data, they just execute the rules as tests.

Property-Based Testing. PBT is a flexible random testing technique that originates from the functional programming community, where it was applied to check algebraic properties of functions-under-test. The strength of this testing technique lies in its flexible generators that can easily be combined or extended and that facilitate the generation of complex test data.

In recent years, PBT was extended with the support for MBT, i.e., it can evaluate state-machine properties, which were explained in Section 2.1. The advantage of the MBT feature of PBT is that it allows for a flexible model definition in a high-level programming language. Unlike PBT, many other MBT approaches require the tester to learn specific modelling languages.

As already explained in Section 2.1, PBT was originally introduced by Claessen and Hughes with a tool called QuickCheck [47]. Various reimplementations followed that are based on the concepts of QuickCheck, e.g., ScalaCheck [136], Hypothesis[18] for Python or FsCheck[19] for .NET, the latter of which works with object-oriented languages (C#), as well as with functional programming languages (F#).

A commercial PBT tool called Quviq QuickCheck was introduced by Hughes [89] for Erlang, and it was targeted towards the needs of the industry. The tool was enhanced with a

[18]`https://pypi.python.org/pypi/hypothesis` (visited on 2018-09-19)
[19]`https://fscheck.github.io/FsCheck` (visited on 2018-09-19)

graphical user interface that enables the creation of models with state-machine diagrams [20]. Moreover, it has been shown that it can be applied for large scale automotive systems [88].

Another PBT tool is called PropEr (Property-based testing for Erlang)[20]. This open-source tool works closely together with Erlang's type system [142]. Moreover, it has an interesting extension for targeted PBT [119], which improves the random test-case generation method of PBT by applying a search strategy.

PBT was already performed for various applications domains, like testing safety-critical software [21, 66, 182] or protocol testing [22, 144]. Furthermore, it is especially suited for testing web-service applications, because it provides a good way to verify that a variety of form inputs are supported without problems. This application domain contains numerous approaches related to our work that are described below.

López et al. [118] presented a domain-specific language (DSL) that allows non-experts to perform automatic test-data generation with QuickCheck. The DSL reuses syntax from the web services description language (WSDL) in order to generate well-formed XML for the input of web services. It supports constraints for different data types and combinators that enable the application of constraints to all kinds of data. The difference between this approach and our work is that it does not consider state machines and that the generator definition must be created manually.

Lampropoulos and Sagonas [109] present a similar approach that automatically reads the WSDL specification of a web service and makes web-service calls with generated data. The approach was implemented with PropEr. They support many data types, but only a few constraints for the data. However, they show how additional constraints can be added manually. In contrast to our work, they also do not use state machines to test the service behaviour. They only test if the web-service result is valid and if no error occurred.

A similar approach was presented by Li et al. [116]. They also show how WSDL can be applied to automatically derive generators, but the focus of their work is primarily on evolving web services. Their approach facilitates adapting the test environment to a new version of a web service. This is achieved by automatically generating refactoring scripts for the evolving test code. The difference to our work is that their models have to be created manually by the user and that their focus lies on evolving web services.

Frelund et al. [67] present a library for testing web services called Jsongen, which can generate data in the JavaScript Object Notation (JSON), i.e., a compact data format. Many web services communicate via JSON because it is a convenient language to encode data. Their library uses JSON schemas with the structure of the data, data types and data constraints to automatically create QuickCheck generators. They apply these QuickCheck generators to produce input data that fulfils the requirements of a web-service call. Their library is evaluated by testing a small service, where users can post questions and answers.

Benac Earle et al. [62] extend this library so that the JSON schema also includes an abstract specification of the service behaviour. This specification is in the form of a finite state machine (FSM). In the previous work, the FSM definition had to be created separately from the JSON schema for the web-service data. In this work, they show how it can be encoded in the JSON schema. Their FSM is defined with hyperlinks that represent the events of the FSM and the states can be chosen dynamically. In contrast to our work, the JSON schema for the service has to be produced manually and it is not part of the system, like our rule engine models. Furthermore, their approach was only evaluated with a small test web service; they have not made a comprehensive case study.

The most similar work to ours was presented by Francisco et al. [64]. They show a framework that automatically derives QuickCheck models from a WSDL description and OCL semantic constraints. They show how the models can be applied to automatically test both

[20] `http://proper.softlab.ntua.gr` (visited on 2018-09-19)

stateless and stateful web services with generated input data. The WSDL description contains information about the required data, the data structures, data types and the possible operations. The OCL constraints define pre- and postconditions for the operations and can describe a state machine for the service behaviour. The used service description is very similar to our business-rule models, but their generators consider only data types, while we also support constraints for the data, like a minimum value for an integer. Another difference is that the OCL semantic constraints are added manually. Our business-rule models were already part of the web-service architecture.

Summary. To the best of our knowledge, there is no other work that uses inherent web-service artefacts, i.e., business-rule models to automatically derive PBT models. Although there are some similar publications that show how PBT models can be used for web services, they mostly rely on a manual specification of a model separate from the web-service implementation. Apart from that, our approach can directly be applied to a system artefact, which is also used directly on the server-side to verify if a command is permitted in the current state and if the attributes are fitting to the model. Furthermore, the other approaches were all implemented with functional programming languages. Our approach uses C# to define the properties in an object-oriented way.

It should be mentioned that it usually does not make sense to generate tests from system artefacts, since there needs to be redundancy for the test oracle. However, in the case that the system artefacts are not perfectly integrated into the SUT, it is worth to apply them for testing. Moreover, such artefacts can be exploited for load testing, where the missing redundancy is not an issue.

8.2 Integrating Statistical Model Checking Into Property-Based Testing

A related technique that can perform similar evaluations as our SMC integration is called statistical software testing [69, 177, 189, 192]. This testing method works with randomly generated inputs that are produced according to certain probability distributions or a specific usage model, like a Markov chain. The generated inputs are given to a software-under-investigation, and a statistical analysis is performed with the intention to find faults or for forecasting faults, e.g., for a reliability assessment of software. In contrast, our combined method can check models as well as (software) systems, and we are able to apply sophisticated SMC algorithms and utilise PBT features, like its powerful generators.

Further related work is also the statistical analysis of black-box probabilistic systems by Younes [196] and Sen et al. [164]. Similar to our work, these approaches support the evaluation of systems that can only be passively observed, and they also apply Monte Carlo simulations and hypothesis testing. However, in contrast to our method, they work with probabilistic systems that do not allow any control over the sample generation, i.e., samples cannot be generated on demand and only a fixed set of samples is available. Additionally, they do not aim at the evaluation of both models and systems.

A related approach that also works with probabilistic models is probabilistic programming [71, 138, 188], which introduces probability distributions into normal programming languages. There exist numerous probabilistic programming languages and probabilistic programming systems that enable the definition of probabilistic models, like Infer.NET [188] from Microsoft or PyMC3 [160] for Python. These models can be applied for different inference techniques, like Bayesian inference [100] or Markov chain Monte Carlo inference [129]. The difference to our approach, or to SMC in general, is that probabilistic programming does not aim to evaluate quantitative properties, but targets probabilistic inference. Most importantly, it does not support PBT.

As already explained, PBT is a flexible random testing technique that facilitates the generation of complex test data. However, it does not support statistical evaluations, except for simple algorithms like Monte Carlo simulation that are already supported by existing PBT tools. For example, with ScalaCheck [136] the required number of samples can be specified and it can report the number of failing samples. This enables a simple Monte Carlo simulation. In contrast, our focus is also on more sophisticated algorithms, like hypothesis testing, but we also apply Monte Carlo methods, because they are common and useful SMC algorithms.

Statistical Model Checking. SMC [2, 115, 198] is an evaluation method that can answer both qualitative and quantitative questions. In order to answer these questions, it includes various algorithms, like Monte Carlo or hypothesis testing methods, which were explained in Section 2.2. SMC was applied in several case studies. Common application areas are the evaluation of protocols [41, 83], biological systems [49, 55] and real-time systems [56, 108]. Moreover, there exist various tools that implement different SMC algorithms and are related to our approach.

A tool that provides similar functionality is UPPAAL-SMC [42]. This tool supports SMC for priced timed automata, which can have weights on transitions and probability distributions for the dwell time in locations. It supports hypothesis testing and probability comparison and estimation by applying Wald's sequential probability ratio test (SPRT) [187] and Monte Carlo simulation with Chernoff-Hoeffding bound [79].

The probabilistic model checker PRISM was also extended with SMC functionality [106]. Similar to UPPAAL-SMC it supports priced timed automata, but it also supports discrete- and continuous-time Markov chains, Markov decision processes and probabilistic automata. They also support the same algorithms as UPPAAL-SMC, i.e., the SPRT and Monte Carlo simulation with Chernoff-Hoeffding bound.

VESTA is another SMC tool that supports hypothesis testing of properties in probabilistic computation tree logic (PCTL) and continuous stochastic logic (CSL) [165]. For modelling, VESTA uses a language, which is related to PRISM in order to specify discrete-time and continuous-time Markov chains. Furthermore, the tool includes an interface to describe models in probabilistic rewrite theories with the algebraic specification language PMAUDE. AlTurki and Meseguer [14] presented an extension of VESTA called PVESTA. This extension includes parallel algorithms for SMC and client-server support.

Another statistical model checker called Ymer was presented by Younes [197]. It is similar to PVESTA and supports properties in PCTL and CSL and uses the SPRT. For modelling, it applies an extension of the PRISM language, which allows for the definition of time-homogeneous generalised semi-Markov processes.

The most similar to our work is from Jegourel et al. [94] and Boyer et al. [38]. First, they had developed the SMC platform PLASMA, which was later replaced by the PLASMA-lab library. The library can perform SMC for multiple modelling languages. For example, it supports the PRISM language and biological languages, it has plugins for Matlab, SystemC and further plugins can be implemented for other modelling languages. This is a nice feature, because it allows for the creation of a custom statistical model checker. However, in order to write a plugin for PLASMA-lab, a user has to be familiar with the architecture of the library and also with the logics for the property definition. The library uses bounded linear temporal logic (BLTL) for the definition of properties and as SMC algorithms it supports simple Monte Carlo, Monte Carlo with Chernoff-Hoeffding bound and SPRT. Furthermore, Legay et al. [114] presented an algorithm for change detection called cumulative sum (CUSUM), which was also added to the PLASMA-lab library.

Summary. Existing SMC tools often have a rather limited modelling language. In order to reduce the effort in modelling and specification an additional layer of abstraction, i.e., "syntactic sugar", can be added. For example, David et al. presented a simulation method for biological systems for UPPAAL-SMC by translating these systems to timed automata [55]. Another approach that enables a high-level specification of Systems of Systems (SoS) and SoS requirements was presented by Arnold et al. [16]. They show how a contract language can be used to define properties, which they translate to BLTL formulas for PLASMA-lab. In contrast, we do not introduce a new language for the model or property definition and hence do not need translators. With C#, we utilise an existing high-level programming language familiar to many developers in the industry. We show that the models and the properties to be checked can be easily defined in an object-oriented programming language. There is no need to learn a new notation or (temporal) logic.

Another advantage is the powerful test-data generators, which are the major ingredient of PBT. These generators can be freely combined and are especially useful for applications, which require a large amount of complex input data, like information systems. Additionally, they support the generation of data with certain probability distributions, which is necessary for stochastic models.

To the best of our knowledge, no existing work combines SMC with PBT, except for papers on PBT tools that report the number of passed and failed test-cases using Monte Carlo simulation.

8.3 Model-Based Prediction and Verification of Performance

A number of related approaches in the area of PBT are concerned with testing concurrent software [19, 36, 48, 87, 90, 137]. For example, Claessen et al. [48] presented a testing method that can find race conditions in Erlang with QuickCheck and a user-level scheduler called PULSE. A similar approach was shown by Norell et al. [137]. They demonstrated an automated way to test blocking operations, i.e., operations that have to wait until a certain condition is met.

Another concurrent PBT approach was demonstrated by Hughes et al. [90]. They showed how PBT can be applied to test distributed file-synchronisation services, like Dropbox. The closest related work we found in this area was from Arts [18]. It shows a load-testing approach with QuickCheck that can run user scenarios on an SUT in order to determine the maximum supported number of users. In contrast to our approach, Arts does not consider stochastic usage profiles and the user scenarios are only tested on an SUT, but not simulated at model-level.

Performance Testing. Related work is also in the area of performance testing [57, 126, 130, 184], which is a class of testing techniques for checking performance requirements that are concerned with responsiveness, resource utilisation, availability, scalability, and reliability of a system. The most common performance testing methods are load testing, where a system is tested under an expected load that is usually achieved by simulating multiple users, and stress testing, which aims to test the upper limits of a system and to find bottlenecks. Moreover, there is a method called soak, endurance, or stability testing that produces expected load over a long period in order to find issues, like memory leaks. Another technique, called volume testing [132] tries to assess if a system supports large volumes of data, e.g., a large database.

There exist various tools for performance testing and load generation [153, 184], which are related to our approach, since they also support the simulation of user populations. For example, Neoload[21] is a performance testing and measurement tool for mobile and web applications that can simulate user populations. A similar open source tool is Apache JMeter [74].

[21] `https://www.neotys.com/neoload/overview` (visited on 2018-09-19)

Initially, it was only built for websites, but since recently it also supports other applications areas. Another tool called LoadRunner [96] from HP supports the simulation of thousands of users and it works for various software platforms, like .NET or Java. The load-testing tool LoadUI[22] has its focus on web-service testing. It enables a flexible test execution that can be modified while running, and it has an interactive graphical interface that supports an easy configuration for the user. The tool was originally open source, but newer versions are only commercially available.

The most related approaches are mainly in the area of load or stress testing. For example, Menascé [127] presented a load testing approach for web sites that works with user interaction scripts to simulate the user behaviour.

A similar approach for benchmarking web servers was illustrated by Banga and Druschel [27]. Their work introduced a new request generation strategy, and they measured the effect of packet losses and network delays on the performance.

Another load-testing method was introduced by Draheim et al. [59]. They showed the simulation of realistic user behaviour with stochastic models and workload models in order to estimate the performance of web applications. A similar approach was presented by Lutteroth and Weber [121].

A related stress testing approach was presented by Krishnamurthy et al. [103]. The work shows a synthetic workload-generation technique that is based on request logs, and should mimic real user behaviour. Moreover, the technique considers inter-request dependencies and is intended for session-based systems. The approach is evaluated with a case study that analysed response times of an e-commerce system.

Another stress testing methodology for finding load-limit points or bottlenecks of game servers was illustrated by Kim et al. [99]. They applied a large number of virtual clients and monitored hardware, network and response times. Their method was evaluated by applying it to a massively multi-player online role-playing game.

In contrast to our work, classical performance or load testing is mostly performed directly on an SUT. With our approach, we want to simulate user populations on the model-level as well.

A related performance testing method with PBT was presented by Handley and Hutton [76]. The work showed the combination of QuickCheck with the Criterion benchmarking library in order to enable a run-time analysis of PBT properties. Moreover, they applied a ridge regression to estimate the time complexity of properties in relation to the input size. In contrast to our work, they do not consider user behaviour and their focus is on time complexity of non-simultaneous run times, but we are interested in expected response times of concurrent requests.

Performance Engineering. The area of performance engineering [149, 167, 168, 193] covers further related work, i.e., various approaches were presented that focus on simulation on model-level in order to predict performance.

For example, Becker et al. [30] presented a prediction method with a Palladio component model, which is a meta-model for component-based software architectures that can include performance indicators. With their method, they predicted response times of an online music repository for concurrent system usage. Moreover, they compared their prediction with measurements from a real system.

Book et al. [37] presented a similar model-based approach for the prediction of response times and communication costs of a web application that is accessed over a mobile channel. Their approach works with dialog flow models and with log data including data volume

[22] `https://www.soapui.org/professional/loadui-pro.html` (visited on 2018-09-19)

and time. They apply it to simulate typical user interaction sequences for different mobile channels.

Lu et al. [120] demonstrated a statistical response-time analysis. Their approach takes response-time samples for the construction of a statistical model that is applied to derive upper bounds for response-time estimates. The work is evaluated with a case study of an industrial robotic control system.

Nourikhah et al. [139] show a model-based forecast method for quality of service values, like response times. The method works with time series models and it considers long-range dependencies in the quality of service data. For the evaluation, they took data from ten real Internet web services in order to build their models, and they made predictions for a forecast horizon of up to 48 hours.

Furthermore, there are various SMC approaches [54, 56, 200] that apply a performance analysis or prediction with stochastic timed automata models.

Most of these approaches only apply a model-based analysis, and do not present an automated technique for the evaluation of their model on an SUT. In contrast, with our method we can perform a model-based prediction, and we can also check the accuracy of our predictions by directly testing an SUT within the same tool.

There are also some approaches or tools that can do both, a simulation with a model and testing an SUT. Balsamo et al. [26] gave an overview of various model-based performance-prediction approaches and tools, and they also categorised the features of these tools. For example, the performance-modelling tool SPE•ED [169] is one of these tools. It works with message sequence charts and supports a model-based simulation, as well as an evaluation of object-oriented systems. Another similar tool is called TwoTowers [33]. It is an open-source software tool that can analyse both functional and performance properties. The tool works with models defined in an architectural description language that is based on stochastic process algebras, and it can perform different analyses, like symbolic model checking or discrete event simulation. A disadvantage of these approaches is that they still require a lot of manual effort, e.g., performance data is often only defined manually, since they are applied in an early phase of the software-development life-cycle. In contrast, we also include an automated approach for response-time learning with linear regression, and we can exploit PBT features, because our approach is realised within a PBT tool.

Statistical Model Checking. In the area of SMC, there are also related tools that support similar performance evaluations as our approach. The most related tool is UPPAAL SMC [42]. Similar to our approach, it provides SMC of priced timed automata, which can simulate user populations. It also supports testing real implementations, but for this a test adapter needs to be implemented, which, e.g., handles the form-data creation. In contrast, we can use PBT features, like data generators, in order to automatically generate form data. In Addition, we can model in a programming language. This helps testers, who are already familiar with this language, as they do not have to learn new notations.

Related approaches also utilise learning techniques. For example, Grinchtein [73] learns time-deterministic event-recording automata via active automata learning, which are similar to our models. The learning method works with membership and equivalence queries that are given to a teacher and it applies observation tables or timed decision trees, which are minimised. A similar approach was presented by Verwer et al. [183]. They passively learn probabilistic real-time automata from positive timed strings by applying statistical state merging and transition splitting. Schmidt et al. [161] also present a learning technique for positive time-labelled data. They learn process models that are probabilistic real-time automata with a state merging method that is based on clustering. In contrast to these learning approaches, we learn response-time distributions and add them to existing automata models, and we present a statistical performance prediction and testing method with SMC.

Deployment Testing. Another domain with related work is deployment testing. For example, various approaches apply a performance analysis of system deployments [122, 154, 199]. However, in contrast to our work, they do not apply a model that is derived from a reference SUT in order to evaluate the performance of SUT deployments under specific usage scenarios.

Evaluation of MQTT. Some other work already showed how MQTT can be tested functionally [151, 173] and even performance analyses have been performed for MQTT. For example, Lee et al. [112] analysed the effects of the message payload size and the quality of service level on the end-to-end delay and the packet loss rate. A similar analysis has been presented by Thangavel et al. [176]. They compared MQTT to another similar protocol, called constrained application protocol (CoAP), and evaluated the delay for different packet loss rates. Collina et al. [51] also compared MQTT to an alternative solution, and they analysed the delay for different subscriber numbers. However, these approaches did not apply a performance model for simulating MQTT under different usage scenarios.

The most similar work to ours that also applied a performance evaluation of MQTT was presented by Houimli et al. [85]. They modelled MQTT with probabilistic timed automata and checked performance properties with UPPAAL SMC. However, they did not validate their model against real implementations, and hence, it did not include real timing behaviour.

Summary. To the best of our knowledge, our work is novel. (1) No other work applies PBT for evaluating stochastic properties of both real systems and stochastic models that include learned response-time distributions. (2) We are the first who apply SMC to the performance analysis of MQTT brokers with learned latency distributions, and who check the results from the model against real MQTT brokers by performing hypothesis testing. (3) No other work performs SMC on a learned timed model of a reference SUT to derive hypotheses that are verified on SUT deployments in order to check, if they provide comparable response times for given usage profiles.

9 Conclusion

This chapter partially contains contents from all our previous publications that were discussed before in Section 1.9 [3, 5, 6, 9, 12, 162].

In this chapter, we give answers to our research questions based on the experiments and results of the previous chapters. We introduced our research questions in Section 1.5, now we explain what was necessary to find answers for these questions. Moreover, we discuss the contributions of this thesis, which consist of several new techniques that support our performance evaluation method. Finally, we conclude the work and present potential future work.

9.1 Research Questions

RQ1: Can business-rule models be applied as test models for property-based testing in order to perform load testing and also to find bugs?

In Chapter 3, we have introduced an automated testing method that works with business-rule models and is realised with a property-based testing (PBT) tool. We have illustrated, how to translate business-rule models to extended finite state machines (EFSMs) in order to use them as a source for test-case generation with PBT. We have formalised the underlying concepts and algorithms of our method and presented an evaluation with an industrial web-service application.

A question that might come up concerns the missing redundancy when generating the test models from the business rules. If a rule engine would be implemented optimally, then our approach would only test the interpreter of the business-rule models. However, in practice programmers often change the source code without considering the rules. Hence, it makes sense to verify that the system-under-test (SUT) still conforms to the model. Especially for custom rule-engine implementations and evolving applications, it is important to test this conformance. Therefore, we have developed an automated approach that verifies this conformance efficiently.

The evaluation has shown that our method is able to reveal bugs and issues that needed to be fixed. Next, we discuss the kind of bugs we found.

RQ1.1: What kind of bugs and issues can be found?

We found eight issues that were confirmed by our industrial partner AVL. The issues concerned the SUT, the testing framework and the business-rule models. In the following, we give an overview of the issues that we were able to find with our testing approach.

The business-rule models partially underspecified the SUT's behaviour and there were deviations of the SUT from the business rules. For example, the business-rule models contained queries for drop down menus that were less strict than the ones of the SUT.

Another issue was that the SUT produced wrong error messages and exceptions in some cases, i.e., an error message contained wrong information.

We found problems with the string handling, e.g., tab characters were inconsistently replaced in the system. Additionally, there were insufficient regular expressions for the input validation, which unintentionally allowed unwanted special characters.

Finally, the testing framework did not support all the functionality of the system. For example, we found a task that worked normally, when it was performed with the graphical user interface, but it was not possible to execute the task with the testing framework. More details about these issues are given in Section 3.6.

RQ1.2: What are the benefits and drawbacks compared to conventional model-based testing?

For conventional model-based testing (MBT), a model is usually created manually, which requires a lot of effort and is prone to error.

With our approach, we can utilise an existing system artefact of the SUT as a source for MBT. Hence, we have the advantage that we can nearly fully automate the testing process and hereby reduce the manual effort. There is only the need for some human interventions, when the business-rule models have underspecified behaviour.

A disadvantage of our approach is that our test oracle is only contained within our business-rule models, and therefore the ability of finding bugs might be limited due to the missing redundancy that was explained before. In contrast to this, for classical MBT the oracle is defined by the user, which allows the detection of a broader range of bugs. However, we have shown that our approach is still able to reveal various bugs and issues. Hence, it makes sense to apply business-rule models for testing, even if only limited types of bugs can be found.

RQ1.3: What are adequate test-case generation strategies for such models?

The business-rule models were translated to EFSMs that were applied for PBT. With PBT, test cases are generated with a random walk on the model. Additionally, we tested a test-case generation strategy that works with model-based mutation testing. We applied MoMuT as an external test-case generator for our PBT approach and combined the generated test sequences with form data that was produced with FsCheck. The evaluation of this approach showed that we can decrease the test-suite size compared to the random strategy, but the generation time was much higher. Hence, this testing strategy would primarily make sense when the test execution is costly and has to be reduced. However, we prefer the random strategy, because the test-case generation is much faster and it can already cover most of the model with just a few test cases. More details about these generation methods are explained in Chapter 3.

Other test-case generation strategies, like search-based or coverage-based methods, might also be suited for business-rule models. The investigation of such methods is a potential topic for future work.

RQ2: Is it possible to perform statistical model checking within a property-based testing tool?

In Chapter 4, we have illustrated that statistical model checking (SMC) algorithms can be integrated into a PBT tool. For this integration, we introduced new SMC properties that take a classical PBT property, parameters for an SMC algorithm and configurations for PBT as input and produce a quantitative or qualitative result. They apply the PBT property in order to produce samples, which are utilised for the evaluation with SMC. We have implemented the SMC properties for commonly used SMC algorithms: for the Monte Carlo simulation (with Chernoff-Hoeffding bound), for the sequential probability ratio test (SPRT) and for the cumulative sum (CUSUM) algorithm.

RQ2.1: What are the differences to conventional statistical model checking?

By repeating case studies from the SMC literature, we have demonstrated that we can perform evaluations like conventional SMC. The major difference is that we do not need a specialised language for the model or property definition, since we can utilise a high level programming language. This is especially helpful for testers from industry, who do not have to learn new notations to apply our method. Moreover, defining the properties in a programming language has the advantage that we can include features, like loop functionality. Conventional temporal logics that are used to define properties often do not support such features. More details about this issue are given in Section 4.4.1.

RQ2.2: What kind of questions can be answered?

With our SMC approach we can evaluate various questions about the probability of properties of both stochastic models and systems, which is also an advantage of our integration, since most other tools do not provide this feature. We have implemented SMC algorithms that allow us to compute the probability of a given property with a required confidence and a maximum error, we can assess which of two given probabilities is closer to the true probability of the property, and we can evaluate if a change in the probability can be detected. Further SMC algorithms might allow us to answer additional types of questions, but in this work our focus was to show the feasibility of an SMC integration into PBT.

RQ3: Can a property-based testing tool be applied to predict the probability that a system satisfies certain response-time thresholds for specific user populations?

We have demonstrated that we can predict the probability of questions about the expected response time of an SUT as described in Section 6.1. In order to simulate a user population, we apply several stochastic timed automata models concurrently. These models include learned response-time distributions and are combined with usage profiles, which represent the behaviour of real users. We apply these models for a Monte Carlo simulation with Chernoff-Hoeffding bound, which allows us to compute the answer for questions, like "What is the probability that the response time of all requests within a task sequence of a fixed length, i.e., a test case, is under a specific threshold for each user within a population?", or "What is the probability that the latency of each interaction of a client within a given MQTT setup is under a certain threshold?".

RQ3.1: What kind of user populations can be simulated?

We have shown that we can simulate user populations of various sizes. The populations consist of users that interact with the system according to given usage profiles, which describe how frequent certain inputs should be performed and also how much time is required for the input. Example usage profiles were illustrated in Section 6.1. Ideally such profiles should be based on recordings of the real usage of a system. Unfortunately, this was not possible in our case. Hence, we created these profiles in cooperation with domain experts.

RQ3.1: How fast is the prediction?

In order to speed up the simulation in comparison to the real execution of the SUT, we applied a virtual time that is a fraction of real time. For our experiments, we used a virtual time of 1/10 of the actual time. This allowed us to run models 10 times as fast as the execution of the SUT. This was especially important, since we applied a Monte Carlo simulation with a high number of samples for the prediction. Concrete run-time examples of our method are presented in Chapter 7.

Note that different virtual time settings are possible, but it is important to select the setting in a way such that the sample-generation time does not negatively influence the simulation. Hence, we can only accomplish a limited speed up by applying a virtual time.

RQ4: Is it possible to verify these predictions about the expected response time by directly testing a system-under-test?

In Section 6.2, we have shown how we can apply hypothesis testing in order to evaluate the predicted probability of our model simulation. We test the prediction power of our model by checking if the probability of the SUT is not much worse and not much higher than the predicted probability of the SUT. Hence, we check if the SUT is at least as good as our model predicted and additionally we test if the SUT is not much better.

RQ4.1: What is an efficient way to test the predictions?

We apply the sequential probability ratio test (SPRT), which is a hypothesis testing method, as explained in Section 2.2.3. This algorithm allows us to stop sampling, when there is enough evidence to decide for a hypothesis. Hence, we need fewer samples as required for the initial Monte Carlo simulation, which is especially important since a direct test of the SUT is costly, when we apply realistic user-input times.

The evaluation has shown that we only need about 276 samples in the worst case and usually less than 50 samples, which is much better than the 1060 samples that we need for a Monte Carlo simulation.

RQ4.2: How accurate are the predictions?

In order to assess the accuracy of our predictions, we have performed two SPRTs, one for checking probability of the SUT is not much worse, and one for testing if it is not much higher. We applied this method in order to evaluate the learned models for the TFMS. The results are illustrated in Section 7.1. The evaluation showed that our predictions were inaccurate in a few cases, i.e., they were either too optimistic or pessimistic. In Section 7.4, we discuss the possible reasons for the limitations of our models. However, our prediction were still accurate in most of the cases and moreover, we were still able to efficiently verify predictions with our proposed method.

9.2 Contributions

Next, we summarise the major contributions of this thesis that were presented in the previous chapter.

A main contribution is a new MBT approach that applies XML business-rule models in the form of EFSMs for PBT. The aim of this method is to perform load testing, but it was also able to find functional bugs. A partial contribution of this approach, was the formalisation of the underlying concepts and algorithms of PBT with EFSMs and of the translation of the business-rule models to EFSM. Moreover, we presented an extended description of our rule-engine driven SUT and evaluated it in an industrial case study.

Another fundamental contribution is the extension of a PBT tool with SMC algorithms. We integrated the SMC algorithms into PBT by introducing SMC properties that enable a statistical evaluation of PBT properties. With this integration, we can apply the modelling notations from PBT and evaluate PBT properties instead of logical formulas that are used in conventional SMC approaches, and we can analyse both stochastic models and systems. Another partial contribution of this extension is the support for a statistical conformance analysis of a stochastic faulty system by comparing it to an ideal model. The approach was demonstrated for the PBT tool FsCheck, and we published the source in order to contribute to the community. Moreover, we illustrated the applicability of this approach by implementing common SMC algorithms, and we repeated evaluations from the SMC literature.

The next essential contribution is our method of extending a functional model with learned non-functional aspects in order to perform SMC. We have shown how we can apply model-based testing to produce log data that includes response times of simultaneous requests. Based on this data, we learned response-time distributions via linear regression and integrated them into our functional model. Hence, we have illustrated an automated method that enables the enhancement of functional models with non-functional timing aspects.

Building upon our extended functional models, the next central contribution is a model-based simulation method that can predict the expected performance of a system. We demonstrated that these models can be applied to evaluate the response time of different usage scenarios. The evaluation is performed with a Monte Carlo simulation, which allows us to find approximate answers to questions about the expected performance, like "How likely will the system satisfy certain response-time thresholds?".

Moreover, another major contribution is an efficient evaluation method that can test the accuracy of our computed model predictions with hypothesis testing. We apply the sequential probability ratio test to check if probabilities that were computed with a model simulation are close to the true probabilities of the SUT. With this prediction and evolution approach, we aim to maximise the user satisfaction by identifying usage scenarios that show a poor performance.

Finally, the last main contribution is the evaluation of our performance prediction and testing approach with two case studies. We applied our method to an industrial web-service application (the TFMS) and also to IoT protocol implementations (for MQTT). An additional interesting contribution of the evaluation is a new application possibility of our method for deployment testing. We demonstrated that we can apply computed hypotheses from a reference system in order to test the performance of various system deployments with different hardware or network settings.

9.3 Conclusions

In order to conclude this work, we come back to the thesis statements that were introduced in Section 1.7 and explain why we think that they are valid. The thesis statements were the following:

1. The application of business-rule models for model-based testing makes sense for finding bugs and also for load testing. It also supports a higher degree of automation since no manual model definition is needed, like it is usually the case for model-based testing.
2. The application of a functional model for model-based testing enables the extension of the functional model to a model with non-functional behaviour. This can, e.g., be done by learning non-functional aspects, like the response time, from log data collected during the execution of model-based testing.
3. Such an extended model enables a prediction of non-functional properties with a Monte Carlo simulation and these predictions can be efficiently verified with hypothesis testing, since this usually can be done with fewer samples.

We believe that these statements were already supported by the previous sections of this chapter, but we want to highlight the most important points once again.

In order to substantiate the first statement, we have developed an automatic test-case generation approach for business-rule models of a web-service application. We applied this approach to an industrial case study and showed that we can find bugs and that it enables capturing of log data for load testing. In our opinion, this approach makes sense, because it can be nearly fully automated, it can perform important consistency checks, and it facilitates the generation of load data.

For the second statement, we have demonstrated that we can enhance a functional model with response-time distributions, which were learned form log data that we obtained from model-based testing. This method is useful, because it can further exploit already existing functional models for the evaluation of non-functional properties, like performance.

In order to support the third statement, we have introduced a simulation method that applies a timed model for predicting the expected system response-times for users. Moreover, we have shown how such predictions can efficiently be evaluated with hypothesis testing. We believe that this method is helpful, because it enables a fast performance prediction for various usage scenarios, and because it can reduce the testing time that is needed on the real system.

To sum up, this thesis presented various novel techniques, and we have evaluated their usefulness with several case studies. The results were promising. We demonstrated that we can efficiently test functional and non-functional properties of industrial systems. Additionally, our techniques support a high degree of automation and are applicable for testers from industry, since they can be performed with common programming languages.

Finally, we are happy to report that we received positive feedback from our industrial partner AVL. They are pleased with our developed techniques and will integrate them into their regular test cycles. Moreover, it is planned that our methods will be further applied and extended in future projects.

9.4 Future Work

In this section, we will describe potential future work that could be applied to further improve the methods that were discusses in this thesis. We limit this description to the following points that we consider the most promising.

An interesting extension for our testing method with business-rule models that might enhance the bug detection capability is fuzzing [171]. In order to apply this technique, it is necessary to produce inputs and test data that are invalid according to the business-rule models. For example, we could test tasks that are not enabled in the current system state or produce form data that does not meet the restrictions of the business-rule models, like values that are larger than an allowed maximum value. This extension could be realised by introducing custom generators for invalid data and it might reveal faults that are caused by a wrong implementation of the business-rule models. However, our existing approach was still effective for finding bugs and our focus was on producing load data. Hence, we were more interested in producing valid data.

Furthermore, a potential topic for future work is another comparison with a different test-case generation strategy for our business-rule models. In this thesis, we have evaluated the default random generation approach and compared it to a generation strategy that is based on model-based mutation testing. The benefits of the mutation-based approach were that the test suite size could be reduced, but the generation time was much higher. However, other generation strategies might overcome this drawback and provide other advantages.

Another enhancement that can be implemented for our SMC integration are additional SMC algorithms. There are some other algorithms, like an alternative hypothesis testing method [163], that support other application areas or have some advantages compared to the conventional algorithms. The investigation of such methods would make sense, because it might increase the efficiency or enlarge the scope of applications. With our SMC approach, new algorithms can be easily integrated into a PBT tool by introducing a new SMC property.

An improvement that is possible for our performance evaluation method is the integration of all steps of the process into one tool. Now, we nearly perform all the steps in C# and FsCheck. Only the learning phase is done externally with a different tool. Integrating this phase into our C# tool could further increase the usability and efficiency of our approach. It would reduce the communication effort between the tools, like parsing the regression models, and make the application of our approach easier, since no additional tool would need to be installed.

Alternative learning methods [78, 190] are also promising future work. Since linear regression still requires high manual effort, e.g., for feature engineering or data preprocessing, it might make sense to apply other learning methods that allow a higher degree of automation. Moreover, they could also help to further improve the accuracy of our performance prediction. However, for our approach it is important that the learning method produces a fast prediction in order support our simulation that is accelerated by using a virtual time, i.e., a fraction of real time. Hence, the selection of the learning algorithms is limited to methods that fulfil this requirement. We are currently in the process of investigating if neural networks can be applied within our performance evaluation approach.

Another interesting extension of our method, is an analysis of the applicability for performance indicators other than response times, e.g., for energy consumption. In this work, we only illustrated the evaluation of timing aspects, but in principle our method can also be applied for other non-functional properties or costs. In order to perform this extension, it would be necessary to record the desired properties in the log files and it may also be necessary to implement a different integration of the learned results into the functional model.

In summary, it can be said that this thesis presented several novel contributions to the field of property-based testing and statistical model checking. However, there is still potential for improvements and extensions. We are currently working on some of these points, and we hope that also other researchers will take up our ideas and continue our research.

Bibliography

[1] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. "Business process management: A survey". In: *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings*. Vol. 2678. Lecture Notes in Computer Science. Springer, 2003, pp. 1–12. DOI: 10.1007/3-540-44895-0_1 (cit. on p. 26).

[2] Gul Agha and Karl Palmskog. "A survey of statistical model checking". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28.1 (2018), 6:1–6:39. DOI: 10.1145/3158668 (cit. on pp. 1, 2, 16, 101).

[3] Bernhard K. Aichernig and Richard Schumi. "How fast is MQTT? - statistical model checking and testing of IoT protocols". In: *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings*. Vol. 11024. Lecture Notes in Computer Science. Springer, 2018, pp. 36–52. DOI: 10.1007/978-3-319-99154-2_3 (cit. on pp. 10, 51, 65, 73, 83, 97, 107).

[4] 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. 9, 10, 25).

[5] Bernhard K. Aichernig and Richard Schumi. "Property-based testing of web services by deriving properties from business-rule models". In: *Software & Systems Modeling* (Dec. 2017). ISSN: 1619-1374. DOI: 10.1007/s10270-017-0647-0 (cit. on pp. 2, 4, 10, 13, 25, 97, 107).

[6] Bernhard K. Aichernig and Richard Schumi. "Statistical model checking meets property-based testing". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 390–400. DOI: 10.1109/ICST.2017.42 (cit. on pp. 2, 9, 13, 51, 97, 107).

[7] Bernhard K. Aichernig and Richard Schumi. "Towards integrating statistical model checking into property-based testing". In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016*. IEEE, 2016, pp. 71–76. DOI: 10.1109/MEMCOD.2016.7797748 (cit. on pp. 9, 51).

[8] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. "Killing strategies for model-based mutation testing". In: *Journal of Software Testing, Verification and Reliability (STVR)* 25.8 (2015), pp. 716–748. DOI: 10.1002/stvr.1522 (cit. on p. 47).

[9] Bernhard K. Aichernig, Priska Bauerstätter, Elisabeth Jöbstl, Severin Kann, Robert Korošec, Willibald Krenn, Cristinel Mateis, Rupert Schlick, and Richard Schumi. "Learning and statistical model checking of system response times". In: *Software Quality Journal* (Dec. 2017). Accepted with minor revisions. (cit. on pp. 4, 9, 10, 13, 51, 65, 73, 83, 97, 107).

[10] 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. 10, 25, 47, 48).

[11] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, and Stefan Tiran. "Require, test, and trace IT". In: *International Journal on Software Tools for Technology Transfer (STTT)* 19.4 (2017), pp. 409–426. DOI: 10.1007/s10009-016-0444-z (cit. on p. 97).

[12] Bernhard K. Aichernig, Severin Kann, and Richard Schumi. "Statistical model checking of response times for different system deployments". In: *Dependable Software Engineering. Theories, Tools, and Applications - 4th International Symposium, SETTA 2018, Beijing, China, September 4-6, 2018, Proceedings*. Vol. 10998. Lecture Notes in Computer Science. Springer, 2018, pp. 153–169. DOI: 10.1007/978-3-319-99933-3_11 (cit. on pp. 10, 65, 73, 83, 97, 107).

[13] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. "Time for mutants - model-based mutation testing with timed automata". In: *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*. Vol. 7942. Lecture Notes in Computer Science. Springer, 2013, pp. 20–38. DOI: 10.1007/978-3-642-38916-0_2 (cit. on p. 97).

[14] Musab AlTurki and José Meseguer. "PVESTA: a parallel statistical model checking and quantitative analysis tool". In: *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*. Vol. 6859. Lecture Notes in Computer Science. Springer, 2011, pp. 386–392. DOI: 10.1007/978-3-642-22944-2_28 (cit. on p. 101).

[15] Rajeev Alur and David L. Dill. "A theory of timed automata". In: *Theoretical Computer Science (TCS)* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8 (cit. on pp. 22, 23).

[16] Alexandre Arnold, Benoît Boyer, and Axel Legay. "Contracts and behavioral patterns for SoS: the EU IP DANSE approach". In: *Proceedings 1st Workshop on Advances in Systems of Systems, AiSoS 2013, Rome, Italy, 16th March 2013*. Vol. 133. EPTCS. Open Publishing Association, 2013, pp. 47–66. DOI: 10.4204/EPTCS.133.6 (cit. on pp. 60, 102).

[17] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. "Modbat: A model-based API tester for event-driven systems". In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. Vol. 8244. Lecture Notes in Computer Science. Springer, 2013, pp. 112–128. DOI: 10.1007/978-3-319-03077-7_8 (cit. on p. 97).

[18] Thomas Arts. "On shrinking randomly generated load tests". In: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014*. ACM, 2014, pp. 25–31. DOI: 10.1145/2633448.2633452 (cit. on p. 102).

[19] Thomas Arts, John Hughes, Ulf Norell, Nicholas Smallbone, and Hans Svensson. "Accelerating race condition detection through procrastination". In: *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. ACM, 2011, pp. 14–22. DOI: 10.1145/2034654.2034659 (cit. on p. 102).

[20] Thomas Arts, Kirill Bogdanov, Alex Gerdes, and John Hughes. "Graphical editing support for QuickCheck models". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–6. DOI: 10.1109/ICSTW.2015.7107473 (cit. on p. 99).

[21] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. "Testing AUTOSAR software with QuickCheck". In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–4. DOI: 10.1109/ICSTW.2015.7107466 (cit. on p. 99).

[22] 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*. ACM, 2006, pp. 2–10. DOI: `10.1145/1159789.1159792` (cit. on pp. 2, 99).

[23] James Aspnes and Maurice Herlihy. "Fast randomized consensus using shared memory". In: *Journal of Algorithms* 11.3 (1990), pp. 441–461. DOI: `10.1016/0196-6774(90)90021-6` (cit. on p. 60).

[24] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. "WSDL-based automatic test case generation for web services testing". In: *2005 IEEE International Workshop on Service-Oriented System Engineering (SOSE 2005), 20-21 October 2005, Beijing, China*. IEEE Computer Society, 2005, pp. 207–212. DOI: `10.1109/SOSE.2005.43` (cit. on p. 98).

[25] Paolo Ballarini, Nathalie Bertrand, András Horváth, Marco Paolieri, and Enrico Vicario. "Transient analysis of networks of stochastic timed automata using stochastic state classes". In: *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Vol. 8054. Lecture Notes in Computer Science. Springer, 2013, pp. 355–371. DOI: `10.1007/978-3-642-40196-1_30` (cit. on pp. 8, 23).

[26] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. "Model-based performance prediction in software development: A survey". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310. DOI: `10.1109/TSE.2004.9` (cit. on p. 104).

[27] Gaurav Banga and Peter Druschel. "Measuring the capacity of a web server under realistic loads". In: *World Wide Web* 2.1-2 (1999), pp. 69–83. DOI: `10.1023/A:1019292504731` (cit. on p. 103).

[28] Andrew Banks and Rahul Gupta. *MQTT version 3.1.1.* OASIS Standard. Dec. 2014. URL: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html` (visited on 2018-09-19) (cit. on pp. 5, 66).

[29] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. "WS-TAXI: A WSDL-based testing tool for web services". In: *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*. IEEE Computer Society, 2009, pp. 326–335. DOI: `10.1109/ICST.2009.28` (cit. on p. 98).

[30] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22. DOI: `10.1016/j.jss.2008.03.066` (cit. on p. 103).

[31] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. "A tutorial on uppaal". In: *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*. Vol. 3185. Lecture Notes in Computer Science. Springer, 2004, pp. 200–236. DOI: `10.1007/978-3-540-30080-9_7` (cit. on p. 22).

[32] Axel Belinfante. "Jtorx: A tool for on-line model-driven test derivation and execution". In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 266–270. DOI: `10.1007/978-3-642-12002-2_21` (cit. on p. 97).

[33] Marco Bernardo, Rance Cleaveland, Steve Sims, and W. Stewart. “Twotowers: A tool integrating functional and performance analysis of concurrent systems”. In: *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE XI / PSTV XVIII'98, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), 3-6 November, 1998, Paris, France*. Vol. 135. IFIP Conference Proceedings. Kluwer, 1998, pp. 457–467 (cit. on p. 104).

[34] Mark R. Blackburn, Robert Busser, Aaron Nauman, and Travis R. Morgan. “Model-based testing in practice”. In: *Informatik 2006 - Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*. Vol. 94. LNI. Gesellschaft für Informatik, 2006, pp. 197–203 (cit. on p. 97).

[35] Lynne Blair, Trevor Jones, and Gordon S. Blair. “Stochastically enhanced timed automata”. In: *Formal Methods for Open Object-Based Distributed Systems IV, IFIF TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), September 6-8, 2000, Stanford, California, USA*. Vol. 177. IFIP Conference Proceedings. Kluwer, 2000, pp. 327–347. DOI: `10.1007/978-0-387-35520-7_17` (cit. on p. 23).

[36] Joseph Blomstedt. “Hansei: property-based development of concurrent systems”. In: *Proceedings of the Eleventh ACM SIGPLAN Erlang Workshop, Copenhagen, Denmark, September 14, 2012*. ACM, 2012, pp. 73–80. DOI: `10.1145/2364489.2364505` (cit. on p. 102).

[37] Matthias Book, Volker Gruhn, Malte Hülder, André Köhler, and Andreas Kriegel. “Cost and response time simulation for web-based applications on mobile channels”. In: *Proceedings Fifth International Conference on Quality Software (QSIC 2005), 19–20 September 2005, Melbourne, Australia*. IEEE Computer Society, 2005, pp. 83–90. DOI: `10.1109/QSIC.2005.21` (cit. on p. 103).

[38] Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. “PLASMA-lab: a flexible, distributable statistical model checking library”. In: *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Vol. 8054. Lecture Notes in Computer Science. Springer, 2013, pp. 160–164. DOI: `10.1007/978-3-642-40196-1_12` (cit. on pp. 16, 51, 101).

[39] Eckard Bringmann and Andreas Krämer. “Systematic testing of the continuous behavior of automotive systems”. In: *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems, Shanghai, China, May 20-28, 2006*. ACM, 2006, pp. 13–20. ISBN: 1-59593-402-2. DOI: `10.1145/1138474.1138479` (cit. on p. 98).

[40] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, eds. *Model-Based Testing of Reactive Systems, Advanced Lectures*. Vol. 3472. Lecture Notes in Computer Science. Springer, 2005. ISBN: 3-540-26278-4. DOI: `10.1007/b137241` (cit. on p. 97).

[41] Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. “Checking and distributing statistical model checking”. In: *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, pp. 449–463. DOI: `10.1007/978-3-642-28891-3_39` (cit. on pp. 16, 101).

[42] Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Marius Mikucionis, Danny Bøgsted Poulsen, Axel Legay, and Zheng Wang. "UPPAAL-SMC: statistical model checking for priced timed automata". In: *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012, Tallinn, Estonia, 31 March and 1 April 2012.* Vol. 85. EPTCS. Open Publishing Association, 2012, pp. 1–16. DOI: 10.4204/EPTCS.85.1 (cit. on pp. 16, 51, 101, 104).

[43] 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.* ACM, 2004, pp. 30–38. ISBN: 1-58113-871-7. DOI: 10.1145/1035167.1035173 (cit. on p. 26).

[44] Taolue Chen, Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre. "Quantitative model checking of continuous-time Markov chains against timed automata specifications". In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA.* IEEE Computer Society, 2009, pp. 309–318. DOI: 10.1109/LICS.2009.21 (cit. on p. 23).

[45] Kwang Ting Cheng and A. 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.* Dallas, Texas, USA: ACM Press, 1993, pp. 86–91. ISBN: 0-89791-577-1. DOI: 10.1145/157485.164585 (cit. on p. 25).

[46] Tsun S. Chow. "Testing software design modeled by finite-state machines". In: *IEEE Transactions on Software Engineering* 4.3 (1978), pp. 178–187. DOI: 10.1109/TSE.1978.231496 (cit. on p. 44).

[47] 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.* ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266 (cit. on pp. 1, 13, 98).

[48] Koen Claessen, Michal H. Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf T. Wiger. "Finding race conditions in Erlang with QuickCheck and PULSE". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009.* ACM, 2009, pp. 149–160. DOI: 10.1145/1596550.1596574 (cit. on p. 102).

[49] Edmund M. Clarke, James R. Faeder, Christopher James Langmead, Leonard A. Harris, Sumit Kumar Jha, and Axel Legay. "Statistical model checking in biolab: applications to the automated analysis of t-cell receptor signaling pathway". In: *Computational Methods in Systems Biology, 6th International Conference, CMSB 2008, Rostock, Germany, October 12-15, 2008. Proceedings.* Vol. 5307. Lecture Notes in Computer Science. Springer, 2008, pp. 231–250. DOI: 10.1007/978-3-540-88562-7_18 (cit. on pp. 16, 101).

[50] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. "The AETG system: an approach to testing based on combinatiorial design". In: *IEEE Transactions on Software Engineering* 23.7 (1997), pp. 437–444. DOI: 10.1109/32.605761 (cit. on p. 98).

[51] Matteo Collina, Giovanni Emanuele Corazza, and Alessandro Vanelli-Coralli. "Introducing the QEST broker: scaling the IoT by bridging MQTT and REST". In: *23rd IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2012, Sydney, Australia, September 9-12, 2012.* IEEE, 2012, pp. 36–41. DOI: 10.1109/PIMRC.2012.6362813 (cit. on p. 105).

[52] FsCheck Community. *Model-based testing with FsCheck*. URL: https://fscheck.github.io/FsCheck/StatefulTesting.html (visited on 2018-09-19) (cit. on pp. 14, 15).

[53] Corinna Cortes, Mehryar Mohri, Michael Riley, and Afshin Rostamizadeh. "Sample selection bias correction theory". In: *Algorithmic Learning Theory, 19th International Conference, ALT 2008, Budapest, Hungary, October 13-16, 2008. Proceedings*. Vol. 5254. Lecture Notes in Computer Science. Springer, 2008, pp. 38–53. DOI: 10.1007/978-3-540-87987-9_8 (cit. on pp. 68, 95).

[54] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, Jonas van Vliet, and Zheng Wang. "Statistical model checking for networks of priced timed automata". In: *Formal Modeling and Analysis of Timed Systems - 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*. Vol. 6919. Lecture Notes in Computer Science. Springer, 2011, pp. 80–96. DOI: 10.1007/978-3-642-24310-3_7 (cit. on p. 104).

[55] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, and Sean Sedwards. "Statistical model checking for biological systems". In: *International Journal on Software Tools for Technology Transfer (STTT)* 17.3 (2015), pp. 351–367. DOI: 10.1007/s10009-014-0323-4 (cit. on pp. 16, 101, 102).

[56] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. "Time for statistical model checking of real-time systems". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 349–355. DOI: 10.1007/978-3-642-22110-1_27 (cit. on pp. 61, 101, 104).

[57] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. "Early performance testing of distributed software applications". In: *Proceedings of the Fourth International Workshop on Software and Performance, WOSP 2004, Redwood Shores, California, USA, January 14-16, 2004*. ACM, 2004, pp. 94–103. DOI: 10.1145/974044.974059 (cit. on p. 102).

[58] 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, Atlanta, Georgia, November 5, 2007*. ACM, 2007, pp. 31–36. ISBN: 978-1-59593-880-0. DOI: 10.1145/1353673.1353681 (cit. on p. 97).

[59] Dirk Draheim, John C. Grundy, John G. Hosking, Christof Lutteroth, and Gerald Weber. "Realistic load testing of web applications". In: *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006) Bari, Italy, 22-24 March 2006*. IEEE, 2006, pp. 57–70. DOI: 10.1109/CSMR.2006.43 (cit. on pp. 2, 103).

[60] Marie Duflot, Marta Kwiatkowska, Gethin Norman, and David Parker. "A formal analysis of Bluetooth device discovery". In: *International Journal on Software Tools for Technology Transfer (STTT)* 8.6 (Oct. 2006), pp. 621–632. ISSN: 1433-2779. DOI: 10.1007/s10009-006-0014-x (cit. on p. 61).

[61] Winfried Dulz and Fenhua Zhen. "Matelo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3". In: *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*. IEEE Computer Society, 2003, pp. 336–342. DOI: 10.1109/QSIC.2003.1319119 (cit. on p. 98).

[62] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz-Nieva, and Julio Mariño. "Jsongen: a QuickCheck based library for testing JSON web services". In: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014*. ACM, 2014, pp. 33–41. ISBN: 978-1-4503-3038-1. DOI: 10.1145/2633448.2633454 (cit. on pp. 6, 99).

[63] Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. "Model-based, mutation-driven test case generation via heuristic-guided branching search". In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*. ACM, 2017, pp. 56–66. DOI: 10.1145/3127041.3127049 (cit. on p. 48).

[64] Miguel A. Francisco, Macías López, 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*. ACM, 2013, pp. 79–86. ISBN: 978-1-4503-2385-7. DOI: 10.1145/2505305.2505306 (cit. on pp. 6, 25, 99).

[65] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. "Does automated unit test generation really help software testers? A controlled empirical study". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.4 (2015), 23:1–23:49 (cit. on p. 1).

[66] Lars-Åke Fredlund, Ángel Herranz-Nieva, and Julio Mariño. "Applying property-based testing in teaching safety-critical system programming". In: *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2015, Madeira, Portugal, August 26-28, 2015*. IEEE Computer Society, 2015, pp. 309–316. DOI: 10.1109/SEAA.2015.53 (cit. on p. 99).

[67] 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 pp. 2, 99).

[68] Jiro Fujita, Dmitry Arkhipkin, Michael Cherney, and Jerome Lauret. "Development of MQTT-channel access bridge". In: *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017): Barcelona, Spain, October 8-13, 2017*. JACoW Publishing, 2018, pp. 1916–1918. DOI: 10.18429/JACoW-ICALEPCS2017-THPHA198 (cit. on p. 6).

[69] Stewart N Gardiner. "Statistical software testing". In: *Testing Safety-Related Software*. Springer, 1999, pp. 155–170 (cit. on p. 100).

[70] Robert L Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002 (cit. on p. 1).

[71] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. "Probabilistic programming". In: *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. ACM, 2014, pp. 167–181. DOI: 10.1145/2593882.2593900 (cit. on p. 100).

[72] Zakkula Govindarajulu. *Sequential Statistics*. World Scientific, 2004. ISBN: 978-981-238-905-3 (cit. on p. 17).

[73] Olga Grinchtein. "Learning of Timed Systems". PhD thesis. Uppsala University, Sweden, 2008 (cit. on p. 104).

[74] Emily H Halili. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008 (cit. on p. 102).

[75] Grégoire Hamon, Leonardo De Moura, and John Rushby. *Automated test generation with SAL*. Tech. rep. Computer Science Laboratory, SRI International, 2005 (cit. on p. 98).

[76] Martin A. T. Handley and Graham Hutton. "AutoBench: comparing the time performance of Haskell programs". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, St. Louis, MO, USA, September 27-28, 2018*. ACM, 2018, pp. 26–37. ISBN: 978-1-4503-5835-4. DOI: 10.1145/3242744.3242749 (cit. on p. 103).

[77] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9 (cit. on p. 36).

[78] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer, 2009. ISBN: 9780387848570 (cit. on pp. 21, 69, 70, 113).

[79] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. "Approximate probabilistic model checking". In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 73–84. DOI: 10.1007/978-3-540-24622-0_8 (cit. on p. 101).

[80] Holger Herbst. *Business Rule-oriented Conceptual Modeling*. Physica-Verlag HD, 1997. ISBN: 978-3-642-59260-7. DOI: 10.1007/978-3-642-59260-7 (cit. on p. 26).

[81] Anders Hessel and Paul Pettersson. "Cover-a test-case generation tool for timed systems". In: *Testing of Software and Communicating Systems: Work-in Progress and Position Papers, Tool Demonstrations, and Tutorial Abstracts of TestCom/FATES 2007, Tallinn, Estonia June 26-29, 2007*. Mälardalen University Embedded Systems Institute, 2007, pp. 31–34 (cit. on p. 97).

[82] Wassily Hoeffding. "Probability inequalities for sums of bounded random variables". In: *Journal of the American Statistical Association* 58.301 (1963), pp. 13–30. ISSN: 01621459 (cit. on p. 16).

[83] Peter Höfner and Annabelle McIver. "Statistical model checking of wireless mesh routing protocols". In: *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 322–336. DOI: 10.1007/978-3-642-38088-4_22 (cit. on pp. 16, 101).

[84] Hans-Martin Hörcher and Jan Peleska. "Using formal specifications to support software testing". In: *Software Quality Journal* 4.4 (1995), pp. 309–327. DOI: 10.1007/BF00402650 (cit. on p. 97).

[85] M. Houimli, L. Kahloul, and S. Benaoun. "Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things". In: *2017 International Conference on Mathematics and Information Technology (ICMIT), December 04-05 2017, Adrar, Algeria*. IEEE, Dec. 2017, pp. 214–221. DOI: 10.1109/MATHIT.2017.8259720 (cit. on p. 105).

[86] John A Hoxmeier and Chris DiCesare. "System response time and user satisfaction: an experimental study of browser-based applications". In: *Proceedings of the Association of Information Systems Americas Conference (AMCIS), New Orleans, Louisiana, USA, August 16-18, 2018*. Vol. 347. AIS Electronic Library (AISeL), 2000, pp. 140–145. URL: http://aisel.aisnet.org/amcis2000/347 (visited on 2018-09-19) (cit. on p. 1).

[87] John M. Hughes and Hans Bolinder. "Testing a database for race conditions with QuickCheck". In: *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. ACM, 2011, pp. 72–77. DOI: `10.1145/2034654.2034667` (cit. on p. 102).

[88] John Hughes. "Experiences with QuickCheck: testing the hard stuff and staying sane". In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Vol. 9600. Lecture Notes in Computer Science. Springer, 2016, pp. 169–186. DOI: `10.1007/978-3-319-30936-1_9` (cit. on p. 99).

[89] 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*. Vol. 4354. Lecture Notes in Computer Science. Springer, 2007, pp. 1–32. DOI: `10.1007/978-3-540-69611-7_1` (cit. on pp. 13, 98).

[90] 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. 102).

[91] 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*. Vol. 4581. Lecture Notes in Computer Science. Springer, 2007, pp. 1–12. DOI: `10.1007/978-3-540-73066-8_1` (cit. on p. 97).

[92] Jacques Janssen and Raimondo Manca. *Applied semi-Markov processes*. Springer Science & Business Media, 2006. ISBN: 9780387295480 (cit. on p. 23).

[93] Claude Jard and Thierry Jéron. "TGV: theory, principles and algorithms". In: *International Journal on Software Tools for Technology Transfer (STTT)* 7.4 (2005), pp. 297–315. DOI: `10.1007/s10009-004-0153-x` (cit. on p. 97).

[94] Cyrille Jégourel, Axel Legay, and Sean Sedwards. "A platform for high performance statistical model checking - PLASMA". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 498–503. DOI: `10.1007/978-3-642-28756-5_37` (cit. on pp. 16, 51, 101).

[95] Sumit Kumar Jha, Edmund M. Clarke, Christopher James Langmead, Axel Legay, André Platzer, and Paolo Zuliani. "A bayesian approach to model checking biological systems". In: *Computational Methods in Systems Biology, 7th International Conference, CMSB 2009, Bologna, Italy, August 31-September 1, 2009. Proceedings*. Vol. 5688. Lecture Notes in Computer Science. Springer, 2009, pp. 218–234. DOI: `10.1007/978-3-642-03845-7_15` (cit. on p. 16).

[96] Cao Jinyuan. "The application of Load Runner in software performance test". In: *Computer Development & Applications* 5 (2012), p. 014 (cit. on p. 103).

[97] Abdul Salam Kalaji, Robert M. Hierons, and Stephen Swift. "Generating feasible transition paths for testing from an extended finite state machine (EFSM)". In: *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*. IEEE Computer Society, 2009, pp. 230–239. DOI: `10.1109/ICST.2009.29` (cit. on p. 29).

[98] Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. "Formal analysis and testing of real-time automotive systems using UPPAAL tools". In: *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*. Vol. 9128. Lecture Notes in Computer Science. Springer, 2015, pp. 47–61. DOI: 10.1007/978-3-319-19458-5_4 (cit. on p. 97).

[99] Ju Young Kim, Jin Ryong Kim, and Chang Joon Park. "Methodology for verifying the load limit point and bottle-neck of a game server using the large scale virtual clients". In: *10th International Conference on Advanced Communication Technology ICACT, Phoenix Park, Korea, Feb. 17-20, 2008*. Vol. 1. IEEE. 2008, pp. 382–386. DOI: 10.1109/ICACT.2008.4493783 (cit. on p. 103).

[100] Daphne Koller, David A. McAllester, and Avi Pfeffer. "Effective Bayesian inference for stochastic programs". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence AAAI 97 and Ninth Innovative Applications of Artificial Intelligence Conference, IAAI 97, July 27-31, 1997, Providence, Rhode Island.* AAAI Press / The MIT Press, 1997, pp. 740–747 (cit. on p. 100).

[101] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. "Data preprocessing for supervised leaning". In: *International Journal of Computer Science* 1.2 (2006), pp. 111–117 (cit. on p. 68).

[102] 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. 48, 97).

[103] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. "A synthetic workload generation technique for stress testing session-based systems". In: *IEEE Transactions on Software Engineering* 32.11 (2006), pp. 868–882. DOI: 10.1109/TSE.2006.106 (cit. on p. 103).

[104] M. Kwiatkowska, G. Norman, and R. Segala. "Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM". In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 194–206. DOI: 10.1007/3-540-44585-4_17 (cit. on p. 60).

[105] M. Kwiatkowska, G. Norman, and D. Parker. "Verifying randomized distributed algorithms with PRISM". In: *Proc. of the Workshop on Advances in Verification, WAVe 2000, Post-Workshop of the Computer Aided Verification Computer: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000*. University of Birmingham, 2000 (cit. on p. 58).

[106] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: verification of probabilistic real-time systems". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47 (cit. on p. 101).

[107] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. "Verifying quantitative properties of continuous probabilistic timed automata". In: *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*. Vol. 1877. Lecture Notes in Computer Science. Springer, 2000, pp. 123–137. DOI: 10.1007/3-540-44618-4_11 (cit. on p. 23).

[108] David Kyle, Jeffery P. Hansen, and Sagar Chaki. "Statistical model checking of distributed adaptive real-time software". In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. Vol. 9333. Lecture Notes in Computer Science. Springer, 2015, pp. 269–274. DOI: 10.1007/978-3-319-23820-3_17 (cit. on p. 101).

[109] Leonidas Lampropoulos and Konstantinos F. 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*. Vol. 98. EPTCS. Open Publishing Association, 2012, pp. 3–16. DOI: 10.4204/EPTCS.98.3 (cit. on pp. 6, 25, 99).

[110] Axel van Lamsweerde. "Formal specification: a roadmap". In: *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*. ACM, 2000, pp. 147–159. DOI: 10.1145/336512.336546 (cit. on p. 97).

[111] Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. "Testing real-time embedded software using UPPAAL-TRON: an industrial case study". In: *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*. ACM, 2005, pp. 299–306. DOI: 10.1145/1086228.1086283 (cit. on p. 97).

[112] Shinho Lee, Hyeonwoo Kim, Dong-kweon Hong, and Hongtaek Ju. "Correlation analysis of MQTT loss and delay according to QoS level". In: *The International Conference on Information Networking 2013, ICOIN 2013, Bangkok, Thailand, January 28-30, 2013*. IEEE Computer Society, 2013, pp. 714–717. DOI: 10.1109/ICOIN.2013.6496715 (cit. on p. 105).

[113] Axel Legay and Sean Sedwards. "On statistical model checking with PLASMA". In: *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*. IEEE Computer Society, 2014, pp. 139–145. DOI: 10.1109/TASE.2014.20 (cit. on pp. 16, 17).

[114] Axel Legay and Louis-Marie Traonouez. "Statistical model checking with change detection". In: *Transactions on Foundations for Mastering Change* 1 (2016), pp. 157–179. DOI: 10.1007/978-3-319-46508-1_9 (cit. on pp. 18, 101).

[115] Axel Legay, Benoît Delahaye, and Saddek Bensalem. "Statistical model checking: an overview". In: *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. Vol. 6418. Lecture Notes in Computer Science. Springer, 2010, pp. 122–135. DOI: 10.1007/978-3-642-16612-9_11 (cit. on pp. 16, 101).

[116] Huiqing Li, Simon Thompson, Pablo Lamela Seijas, and Miguel Angel Francisco. "Automating property-based testing of evolving web services". In: *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*. ACM, 2014, pp. 169–180. ISBN: 978-1-4503-2619-3. DOI: 10.1145/2543728.2543741 (cit. on pp. 6, 99).

[117] Chien-Hung Liu, Shu-Ling Chen, and Xue-Yuan Li. "A WS-BPEL based structural testing approach for web service compositions". In: *The Fourth IEEE International Symposium on Service-Oriented System Engineering, SOSE 2008, 18-19 December 2008, Jhongli, Taiwan*. IEEE Computer Society, 2008, pp. 135–141. DOI: 10.1109/SOSE.2008.30 (cit. on p. 98).

[118] LMC Macıias López, Henrique Ferreiro, and T Arts. “A DSL for web services automatic test data generation”. In: *Draft Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages, Nijmegen, Netherlands, August 28 - 30, 2013*. Radbound University Institute for Computing and Information Sciences, 2013 (cit. on pp. 2, 99).

[119] Andreas Löscher and Konstantinos Sagonas. “Targeted property-based testing”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. ACM, 2017, pp. 46–56. DOI: 10.1145/3092703.3092711 (cit. on p. 99).

[120] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. “A statistical response-time analysis of real-time embedded systems”. In: *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS 2012, San Juan, PR, USA, December 4-7, 2012*. IEEE Computer Society, 2012, pp. 351–362. DOI: 10.1109/RTSS.2012.85 (cit. on p. 104).

[121] Christof Lutteroth and Gerald Weber. “Modeling a realistic workload for performance testing”. In: *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*. IEEE Computer Society, 2008, pp. 149–158. DOI: 10.1109/EDOC.2008.40 (cit. on p. 103).

[122] Haroon Malik and Elhadi M. Shakshuki. “Classification of post-deployment performance diagnostic techniques for large-scale software systems”. In: *The 5th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2014)/ The 4th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2014)/ Affiliated Workshops, September 22-25, 2014, Halifax, Nova Scotia, Canada*. Vol. 37. Procedia Computer Science. Elsevier, 2014, pp. 244–251. DOI: 10.1016/j.procs.2014.08.036 (cit. on p. 105).

[123] Silvio Marcovic. “Integrating an External Test-Case Generator into a Property-Based Testing Tool for Testing Web-Services”. MA thesis. Graz University of Technology, 2017 (cit. on p. 48).

[124] Philip Mayer and Daniel Lübke. “Towards a BPEL unit testing framework”. In: *Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), TAV-WEB 2006, Portland, Maine, USA, July 17, 2006*. ACM, 2006, pp. 33–42. DOI: 10.1145/1145718.1145723 (cit. on p. 98).

[125] P. McDermott-Wells. “What is Bluetooth?” In: *IEEE Potentials* 23.5 (Dec. 2005), pp. 33–35. ISSN: 0278-6648. DOI: 10.1109/MP.2005.1368913 (cit. on p. 61).

[126] J Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications*. Microsoft Press, 2010. ISBN: 9780735646001 (cit. on p. 102).

[127] Daniel A. Menascé. “Load testing of web sites”. In: *IEEE Internet Computing* 6.4 (2002), pp. 70–74. DOI: 10.1109/MIC.2002.1020328 (cit. on pp. 2, 103).

[128] Nikola Milanovic and Miroslaw Malek. “Current solutions for web service composition”. In: *IEEE Internet Computing* 8.6 (2004), pp. 51–59. DOI: 10.1109/MIC.2004.58 (cit. on p. 25).

[129] Brian Milch and Stuart J. Russell. “General-purpose MCMC inference over relational structures”. In: *UAI'06, Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence, Cambridge, MA, USA, July 13-16, 2006*. AUAI Press, 2006 (cit. on p. 100).

[130] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. Theory in practice. O'Reilly Media, 2009. ISBN: 9780596555436 (cit. on pp. 1, 102).

[131] "Mutation-driven test case generation using short-lived concurrent mutants - first results". In: *Computing Research Repository (CoRR)* abs/1601.06974 (2016). Withdrawn. arXiv: 1601.06974 (cit. on p. 48).

[132] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011 (cit. on pp. 1, 102).

[133] Nico JD Nagelkerke. "A note on a general definition of the coefficient of determination". In: *Biometrika* 78.3 (1991), pp. 691–692 (cit. on p. 70).

[134] Suresh Nageswaran. "Test effort estimation using use case points". In: *14th in the continuing series of International Internet & Software Quality Week QW 2001, San Francisco, California, USA, 29 May - 1 June, 2001*. Vol. 6. 2001, pp. 1–6 (cit. on p. 1).

[135] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. RFC Editor, 1984. URL: https://tools.ietf.org/html/rfc896 (visited on 2018-09-19) (cit. on p. 91).

[136] R. Nilsson. *ScalaCheck: The Definitive Guide*. IT Pro. Artima Incorporated, 2014. ISBN: 9780981531694 (cit. on pp. 13, 98, 101).

[137] Ulf Norell, Hans Svensson, and Thomas Arts. "Testing blocking operations with QuickCheck's component library". In: *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013*. ACM, 2013, pp. 87–92. DOI: 10.1145/2505305.2505310 (cit. on p. 102).

[138] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. "R2: an efficient MCMC sampler for probabilistic programs". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* AAAI Press, 2014, pp. 2476–2482. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8192 (visited on 2018-09-19) (cit. on p. 100).

[139] Hossein Nourikhah, Mohammad Kazem Akbari, and Mohammad Kalantari. "Modeling and predicting measured response time of cloud-based web services using long-memory time series". In: *The Journal of Supercomputing* 71.2 (2015), pp. 673–696. DOI: 10.1007/s11227-014-1317-4 (cit. on p. 104).

[140] Bart Orriëns, Jian Yang, and Mike P. Papazoglou. "A framework for business rule driven service composition". In: *Technologies for E-Services, 4th International Workshop, TES 2003, Berlin, Germany, September 8, 2003, Proceedings*. Vol. 2819. Lecture Notes in Computer Science. Springer, 2003, pp. 14–27. DOI: 10.1007/978-3-540-39406-8_2 (cit. on p. 26).

[141] Susan S. Owicki and Leslie Lamport. "Proving liveness properties of concurrent programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 455–495. DOI: 10.1145/357172.357178 (cit. on p. 16).

[142] Manolis Papadakis. "Automatic Random Testing of Function Properties from Specifications". Diploma thesis. National Technical University of Athens, School of Electrical and Computer Engineering, 2010 (cit. on p. 99).

[143] Manolis Papadakis and Konstantinos Sagonas. "A PropEr integration of types and function specifications with property-based testing". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang'11, Tokyo, Japan, September 23, 2011*. Tokyo, Japan: ACM, 2011, pp. 39–50. ISBN: 978-1-4503-0859-5. DOI: 10.1145/2034654.2034663 (cit. on p. 13).

[144] Javier Paris and Thomas Arts. "Automatic testing of TCP/IP implementations using QuickCheck". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang, Edinburgh, Scotland, UK, September 5, 2009*. ACM, 2009, pp. 83–92. DOI: 10.1145/1596600.1596612 (cit. on pp. 2, 99).

[145] Karl Pearson. "Note on regression and inheritance in the case of two parents". In: *Proceedings of the Royal Society of London* 58 (1895), pp. 240–242 (cit. on p. 69).

[146] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on p. 16).

[147] Amir Pnueli and Lenore Zuck. "Verification of multiprocess probabilistic protocols". In: *Distributed Computing* 1.1 (1986), pp. 53–72. ISSN: 1432-0452. DOI: 10.1007/BF01843570 (cit. on p. 58).

[148] Nicolás Poggi, David Carrera, Ricard Gavaldà, Eduard Ayguadé, and Jordi Torres. "A methodology for the evaluation of high response time on e-commerce users and sales". In: *Information Systems Frontiers* 16.5 (2014), pp. 867–885. DOI: 10.1007/s10796-012-9387-4 (cit. on p. 1).

[149] Rob Pooley and Peter King. "The unified modelling language and performance engineering". In: *IEE Proceedings-Software* 146.1 (1999), pp. 2–10 (cit. on p. 103).

[150] Stacy J. Prowell. "JUMBL: A tool for model-based statistical testing". In: *36th Hawaii International Conference on System Sciences (HICSS-36 2003), Proceedings, January 6-9, 2003, Big Island, HI, USA*. IEEE Computer Society, 2003, p. 337. DOI: 10.1109/HICSS.2003.1174916 (cit. on p. 98).

[151] Santiago Hernández Ramos, M. Teresa Villalba, and Raquel Lacuesta. "MQTT security: a novel fuzzing approach". In: *Wireless Communications and Mobile Computing* 2018 (2018). DOI: 10.1155/2018/8261746 (cit. on p. 105).

[152] A.C. Rencher and W.F. Christensen. *Methods of Multivariate Analysis*. third. Wiley Series in Probability and Statistics. John Wiley & Sons, 2012. ISBN: 9781118391679 (cit. on pp. 68, 70).

[153] Rina and Sanjay Tyagi. "A comparative study of performance testing tools". In: *International Journal of Advanced Research in Computer Science and Software Engineering Research* 3.5 (2013), pp. 1300–1307 (cit. on p. 102).

[154] Eduardo Roloff, Matthias Diener, Alexandre Carissimi, and Philippe Olivier Alexandre Navaux. "High performance computing in the cloud: deployment, performance and cost efficiency". In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, CloudCom 2012, Taipei, Taiwan, December 3-6, 2012*. IEEE Computer Society, 2012, pp. 371–378. DOI: 10.1109/CloudCom.2012.6427549 (cit. on p. 105).

[155] F. Rosenberg and S. Dustdar. "Business rules integration in BPEL - a service-oriented approach". In: *Seventh 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. 25, 26).

[156] 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 p. 25).

[157] Ronald G. Ross. *Principles of the Business Rule Approach*. Boston, MA, USA: Addison-Wesley Professional, 2003. ISBN: 978-0201788938. DOI: 10.1016/j.ijinfomgt.2003.12.007 (cit. on p. 26).

[158] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell'08, Victoria, BC, Canada, 25 September 2008*. ACM, 2008, pp. 37–48. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411292 (cit. on p. 13).

[159] Ahmad A. Saifan and Jürgen Dingel. "A survey of using model-based testing to improve quality attributes in distributed systems". In: *Advanced Techniques in Computing Sciences and Software Engineering, Volume II of the proceedings of the 2008 International Conference on Systems, Computing Sciences and Software Engineering (SCSS), part of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering, CISSE 2008, Bridgeport, Connecticut, USA*. Springer, 2008, pp. 283–288. DOI: 10.1007/978-90-481-3660-5_48 (cit. on p. 97).

[160] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. "Probabilistic programming in Python using PyMC3". In: *PeerJ Computer Science* 2 (2016), e55. DOI: 10.7717/peerj-cs.55 (cit. on p. 100).

[161] Jana Schmidt, Asghar Ghorbani, Andreas Hapfelmeier, and Stefan Kramer. "Learning probabilistic real-time automata from multi-attribute event logs". In: *Intelligent Data Analysis* 17.1 (2013), pp. 93–123. DOI: 10.3233/IDA-120569 (cit. on p. 104).

[162] Richard Schumi, Priska Lang, Bernhard K. Aichernig, Willibald Krenn, and Rupert Schlick. "Checking response-time properties of web-service applications under stochastic user profiles". In: *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings*. Vol. 10533. Lecture Notes in Computer Science. Springer, 2017, pp. 293–310. DOI: 10.1007/978-3-319-67549-7_18 (cit. on pp. 10, 65, 72, 73, 97, 107).

[163] Koushik Sen, Mahesh Viswanathan, and Gul Agha. "On statistical model checking of stochastic systems". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 266–280. DOI: 10.1007/11513988_26 (cit. on p. 112).

[164] Koushik Sen, Mahesh Viswanathan, and Gul Agha. "Statistical model checking of black-box probabilistic systems". In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 202–215. DOI: 10.1007/978-3-540-27813-9_16 (cit. on p. 100).

[165] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. "VESTA: a statistical model-checker and analyzer for probabilistic systems". In: *Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*. IEEE Computer Society, 2005, pp. 251–252. DOI: 10.1109/QEST.2005.42 (cit. on p. 101).

[166] Che-Hua Shih, Juinn-Dar Huang, and Jing-Yang Jou. "Stimulus generation for interface protocol verification using the nondeterministic extended finite state machine model". In: *Tenth IEEE International High-Level Design Validation and Test Workshop 2005, Napa Valley, CA, USA, November 30 - December 2, 2005*. IEEE Computer Society, 2005, pp. 87–93. DOI: 10.1109/HLDVT.2005.1568819 (cit. on p. 29).

[167] Connie U. Smith. "Software performance engineering then and now: A position paper". In: *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development, WOSP-C'15, Austin, TX, USA, January 31, 2015*. ACM, 2015, pp. 1–3. DOI: 10.1145/2693561.2693567 (cit. on p. 103).

[168] Connie U. Smith. "Software performance engineering tutorial". In: *16th International Computer Measurement Group Conference, December 10-14, 1990, Orlando, FL, USA, Proceedings*. Computer Measurement Group, 1990, pp. 1311–1318 (cit. on p. 103).

[169] Connie U. Smith and Lloyd G. Williams. "Performance engineering evaluation of object-oriented systems with SPE•ED". In: *Computer Performance Evaluation: Modelling Techniques and Tools, 9th International Conference, St. Malo, France, June 3-6, 1997, Proceedings*. Vol. 1245. Lecture Notes in Computer Science. Springer, 1997, pp. 135–154. DOI: `10.1007/BFb0022203` (cit. on p. 104).

[170] Harry M. Sneed and Shihong Huang. "The design and use of WSDL-Test: a tool for testing web services". In: *Journal of Software Maintenance* 19.5 (2007), pp. 297–314. DOI: `10.1002/smr.354` (cit. on p. 98).

[171] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007. ISBN: 9780321680853 (cit. on p. 112).

[172] Jiliang Tang, Salem Alelyani, and Huan Liu. "Feature selection for classification: A review". In: *Data Classification: Algorithms and Applications*. CRC Press, 2014, pp. 37–64 (cit. on p. 69).

[173] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. "Model-based testing IoT communication via active automata learning". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 276–287. DOI: `10.1109/ICST.2017.32` (cit. on pp. 66, 105).

[174] Robert E Tarjan. *Lecture 10: more Chernoff bounds, sampling, and the Chernoff + Union bound method*. 2009. URL: `http://www.cs.princeton.edu/courses/archive/fall09/cos521/Handouts/probabilityandcomputing.pdf` (visited on 2018-09-19) (cit. on p. 17).

[175] Maurice H. Ter Beek, Antonio Bucchiarone, and Stefania Gnesi. "Formal methods for service composition". In: *Annals of Mathematics, Computing & Teleinformatics* 1.5 (2007), pp. 1–10 (cit. on p. 98).

[176] Dinesh Thangavel, Xiaoping Ma, Alvin C. Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. "Performance evaluation of MQTT and CoAP via a common middleware". In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014*. IEEE, 2014, pp. 1–6. DOI: `10.1109/ISSNIP.2014.6827678` (cit. on p. 105).

[177] Pascale Thévenod-Fosse and Hélène Waeselynck. "An investigation of statistical software testing". In: *Journal of Software Testing, Verification and Reliability (STVR)* 1.2 (1991), pp. 5–25 (cit. on p. 100).

[178] G.J. Tretmans and Hendrik Brinksma. "Côte de resyste – automated model based testing". In: *3rd PROGRESS Workshop on Embedded Systems 2002 - Utrecht, Netherlands, 24 Oct 2002*. STW Technology Foundation, 2002, pp. 246–255. ISBN: 90-73461-34-0 (cit. on p. 97).

[179] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007. ISBN: 978-0-12-372501-1 (cit. on pp. 2, 97).

[180] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Journal of Software Testing, Verification and Reliability (STVR)* 22.5 (2012), pp. 297–312. DOI: `10.1002/stvr.456` (cit. on pp. 2, 3, 97).

[181] 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*. Vol. 4949. Lecture Notes in Computer Science. Springer, 2008, pp. 39–76. DOI: `10.1007/978-3-540-78917-8_2` (cit. on p. 97).

[182] Benjamin Vedder, Henrik Eriksson, Daniel Skarin, Jonny Vinter, and Magnus Jonsson. "Towards collision avoidance for commodity hardware quadcopters with ultrasound localization". In: *2015 International Conference on Unmanned Aircraft Systems (ICUAS) 2015, Denver, Colorado, USA, Jun 9-12, 2015*. IEEE. 2015, pp. 193–203. DOI: 10.1109/ICUAS.2015.7152291 (cit. on p. 99).

[183] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. "A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data". In: *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*. Vol. 6339. Lecture Notes in Computer Science. Springer, 2010, pp. 203–216. DOI: 10.1007/978-3-642-15488-1_17 (cit. on p. 104).

[184] Pallavi M S Vinayak Hegde. "Web performance testing: methodologies, tools and challenges". In: *International Journal of Scientific Engineering and Research (IJSER)* 2 (1 2014). ISSN: 2347–3878 (cit. on p. 102).

[185] Yusuke Wada and Shigeru Kusakabe. "Performance evaluation of a testing framework using QuickCheck and Hadoop". In: *Journal of Information Processing* 20.2 (2012), pp. 340–346. DOI: 10.2197/ipsjjip.20.340 (cit. on p. 13).

[186] Gerd Wagner, Grigoris Antoniou, Said Tabet, and Harold Boley. "The abstract syntax of RuleML - towards a general web rule language framework". In: *2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2004), 20-24 September 2004, Beijing, China*. IEEE Computer Society, 2004, pp. 628–631. DOI: 10.1109/WI.2004.134 (cit. on pp. 27, 98).

[187] Abraham Wald. *Sequential Analysis*. Courier Corporation, 1973 (cit. on pp. 2, 8, 17, 101).

[188] S. S. J. Wang and M. P. Wand. "Using Infer.NET for statistical analyses". In: *The American Statistician* 65.2 (2011), pp. 115–126. DOI: 10.1198/tast.2011.10169 (cit. on p. 100).

[189] Robert J. Weber. "Statistical software testing with parallel modeling: A case study". In: *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*. IEEE Computer Society, 2004, pp. 35–44. DOI: 10.1109/ISSRE.2004.37 (cit. on p. 100).

[190] Brady T. West, Kathleen B. Welch, and Andrzej T. Galecki. *Linear Mixed Models*. CRC Press, 2006. ISBN: 9781420010435 (cit. on pp. 19, 113).

[191] Jodie Wetherall and Steve Woodhead. *Investigation into a modular rule-based testing method for testing business rules in scheduling applications*. 2008. URL: http://gala.gre.ac.uk/2679/ (visited on 2018-09-19) (cit. on p. 98).

[192] James A. Whittaker and Michael G. Thomason. "A Markov chain model for statistical software testing". In: *IEEE Transactions on Software Engineering* 20.10 (1994), pp. 812–824. DOI: 10.1109/32.328991 (cit. on p. 100).

[193] C. Murray Woodside, Greg Franks, and Dorina C. Petriu. "The future of software performance engineering". In: *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*. IEEE Computer Society, 2007, pp. 171–187. DOI: 10.1109/FOSE.2007.32 (cit. on p. 103).

[194] Sewall Wright. "Correlation and causation". In: *Journal of Agricultural Research* 20 (1921), pp. 557–585 (cit. on p. 70).

[195] G. George Yin and Qing Zhang. *Continuous-Time Markov Chains and Applications: A Two-Time-Scale Approach*. Vol. 37. Springer Science & Business Media, 2012. ISBN: 9781461443469 (cit. on p. 23).

[196] Håkan L. S. Younes. "Probabilistic verification for "black-box"systems". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 253–265. DOI: `10.1007/11513988_25` (cit. on p. 100).

[197] Håkan L. S. Younes. "Ymer: a statistical model checker". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 429–433. ISBN: 978-3-540-31686-2. DOI: `10.1007/11513988_43` (cit. on p. 101).

[198] Håkan L. S. Younes and Reid G. Simmons. "Probabilistic verification of discrete event systems using acceptance sampling". In: *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 223–235. DOI: `10.1007/3-540-45657-0_17` (cit. on pp. 16, 101).

[199] Jian Yu, Jun Han, Jean-Guy Schneider, Cameron M. Hine, and Steve Versteeg. "A Petri-net-based virtual deployment testing environment for enterprise software systems". In: *The Computer Journal* 60.1 (2017), pp. 27–44. DOI: `10.1093/comjnl/bxw055` (cit. on p. 105).

[200] Fengling Zhang, Lei Bu, Linzhang Wang, Jianhua Zhao, Xin Chen, Tian Zhang, and Xuandong Li. "Modeling and evaluation of wireless sensor network protocols by stochastic timed automata". In: *Electronic Notes in Theoretical Computer Science* 296 (2013), pp. 261–277. DOI: `10.1016/j.entcs.2013.09.001` (cit. on p. 104).