ABSTRACT

Title of dissertation: MODEL-BASED TESTING OF OFF-NOMINAL BEHAVIORS

Christoph Schulze, Doctor of Philosophy, 2018

Dissertation directed by: Professor Rance Cleaveland
Department of Computer Science

Off-nominal behaviors (ONBs) are unexpected or unintended behaviors that may be exhibited by a system. They can be caused by implementation and documentation errors and are often triggered by unanticipated external stimuli, such as unforeseen sequences of events, out of range data values, or environmental issues. System specifications typically focus on nominal behaviors (NBs), and do not refer to ONBs or their causes or explain how the system should respond to them. In addition, untested occurrences of ONBs can compromise the safety and reliability of a system. This can be very dangerous in mission- and safety-critical systems, like spacecraft, where software issues can lead to expensive mission failures, injuries, or even loss of life. In order to ensure the safety of the system, potential causes for ONBs need to be identified and their handling in the implementation has to be verified and documented.

This thesis describes the development and evaluation of model-based techniques for the identification and documentation of ONBs. Model-Based Testing (MBT) techniques have been used to provide automated support for thorough evaluation of software behavior. In MBT, models are used to describe the system under test (SUT) and to derive test cases for that SUT. The thesis is divided into two parts. The first part develops and evaluates an approach for the automated generation of MBT models and their associated test infrastructure. The test infrastructure is responsible for executing the generated test cases of the models. The models and the test infrastructure are generated from manual test cases for web-based systems, using a set of heuristic transformation rules and leveraging the structured nature of the SUT. This improvement to the MBT process was motivated by three case studies of MBT that we conducted that evaluate MBT in terms of its effectiveness and efficiency for identifying ONBs. Our experience led us to develop automated approaches to model and test-infrastructure creation, since these were some of the most time-consuming tasks associated with MBT.

The second part of the thesis presents a framework and associated tooling for the extraction and analysis of specifications for identifying and documenting ONBs. The framework infers behavioral specifications in the form of system invariants from automatically generated test data using data-mining techniques (e.g. association-rule mining). The framework follows an iterative $test \rightarrow infer \rightarrow instrument \rightarrow retest$ paradigm, where the initial invariants are refined with additional test data. This work shows how the scalability and accuracy of the resulting invariants can be improved with the help of static data- and control-flow analysis. Other improvements include an algorithm that leverages the iterative process to accurately infer invariants from variables with continuous values. Our evaluations of the framework have shown the utility of such automatically generated invariants as a means for

updating and completing system specifications; they also are useful as a means of understanding system behavior including ONBs.

MODEL-BASED TESTING OF OFF-NOMINAL BEHAVIORS

by

Christoph Schulze

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Dedication

To Zamira and Isabella

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and who made my Ph.D. experience one that I will cherish forever.

I would like to express an immeasurable amount of gratitude to my advisor Rance Cleaveland for guiding me through all these years of study. Thank you for being such a great advisor! Thank you also for accommodating my travels to my wife and for letting me work remotely in the last years. You had a great impact on me and my family and all three of us are very grateful for it.

Thank you also to Mikael Lindvall for being a good friend and mentor. Thank you for teaching me how to conduct research studies, write papers, and how to present my work. I am going to miss our discussions!

My appreciation also goes out to my thesis committee. Shuvra Bhattacharyya, Don Perlis and Adam Porter. Thank you for agreeing to serve on my committee and for the hard work reviewing my thesis.

I was very fortunate to be able to work together with many industry partners during my PhD. This work was supported by NASAs Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP). We especially thank Lisa P. Montgomery, Markland Benson, Wesley W. Deadrick, Martha Wetherholt, Sally Godfrey, Art Hughes, Stephen Hard, Lamont Ruley, Dave McComas, Ken Rehm, Ricky Forquer, Robert Wiegand, and Alan Cudmore for their continuous support. Thank you for giving me the opportunity to work on real world problems and for making a small impact on software that is out there right now flying high above us. Thank you also to Daniel Goldman from GlobalNet Services for giving us the opportunity to perform a unique Model-Based Testing study. Thank you also to the NSF and to the CyberCardians for their support of this work.

Thanks to all my coworkers and friends at Fraunhofer. A special thank goes to Mary for keeping us fed and watered. I also want to thank Charles, Madeline, Chris, Michele, Linda, Lucas, Betty, Gudjon, Arnab, and Dharma. Thanks for all the interns that supported my work with the best intern award going to Vignir and Sigurthor. It was a lot of fun working with you! Thanks also to Jenny, Fatima and Jodie at the University of Maryland. Whenever a student needed something you were there for us!

Besides my work I was also very fortunate to meet friends for life here who made College Park really enjoyable. Alex, Kotaro, Brenna, Matt, Varun, Aishwarya, Bhaskar, James, Francisco, Austin, Angjoo, Ran, and Kent! Thank you! Thank you also to the Edmonston House crew Nico, Sam and Tersha for rescuing me out of my weird first room in College Park. It was so much fun living with you guys!

Thank you to my family for their love and most importantly to Zamira the love of my life and my wife. You have always believed in me and supported me and kept me sane, especially when I doubted myself. Your love, support, and sacrifice made this work possible. Without you I could not have done it! Thanks also to our little Isabella who now makes our life even happier and who helped give me the last push I needed to graduate.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ONB	Off-Nominal Behavior
NB	Nominal Behavior
MBD	Model-Based Design
MBT	Model-Based Testing
SUT	System-Under Test
LHS	Left-Hand Side
RHS	Left-Hand Side
FSM	Finite-State Machine
EFSM	Extended Finite-State Machine
GMSEC	Goddard Mission Service Evolution Center
OSAL	Operating System Abstraction Layer
ELEXNET	Electronic Laboratory Exchange Network

Chapter 1: Introduction

Off-nominal behaviors (ONBs) are unexpected or unintended behaviors that may be exhibited by a system [1]. They can be caused by implementation and documentation errors and are often triggered by unanticipated external stimuli, such as unforeseen sequences of events, out-of-range data values, or environmental issues [1,2]. Untested occurrences of ONBs can compromise the safety and reliability of a system, which can lead to expensive mission failures [3] [4], injuries, or even loss of life [5]. In order to ensure the safety of the system, potential causes for ONBs must be identified, and their handling must be specified and verified in the implementation of the system. According to a report about several space craft related incidents by NG Leveson [3], there was *"inadequate emphasis on off-nominal and stress testing. Most software-related accidents have involved situations that were not considered during development or were assumed to be impossible and not handled by the software"*.

System requirements, which typically focus on nominal behavior (NB), often do not address ONBs and how systems should respond to them [6]. Several approaches have been developed to address this problem from a requirements-engineering and safety-engineering perspective [6–9]. These approaches strive to complete the given requirements with respect to the ONBs. Alexander [7] introduced the idea of misuse cases, which are use cases that describe how the system shall react to the occurrence of invalid inputs. Ishimatsu [8] introduces the *System-Theoretic Process Analysis*, a hazard analysis technique that takes off-nominal commands into account. The goal of the technique is the identification of hazardous design flaws such as software and system design errors, and unsafe interaction among system components. Aceituna [9] introduces an approach that formalizes the requirements of a system as a set of concurrent state-machine models, where each model represents a system component. Model checking is then used to identify missing, or invalid behaviors. Day et al. [6] are using SysML [10] to model a system and then perform an automated *Failure Modes and Effects Analysis* [11] to identify causes for ONBs and their potential propagation through the system. These approaches try to analyze ONBs of a system before it is implemented, or a prototype of a system exists. However, potential causes for ONBs can be manifold [12], and it is difficult to identify all potential ONBs and their detrimental effects without the implementation or a prototype of the system.

Model-Based Design (MBD) uses models that abstract information in order to reduce system complexity. This reduction in complexity allows that the resulting models can be used for system verification, code generation and documentation. Model-based testing (MBT) employs the ideas of MBD and applies them to software testing. Instead of generating system code, MBT generates test cases for the system.

This dissertation describes the development and evaluation of model-based techniques to improve the identification and documentation of ONBs. We start by discussing the nature of ONB, specifically their precise definition, how they are caused and what their effects are on a system and its documentation. The rest of the dissertation is then divided into two parts. The first part evaluates MBT to test systems for ONBs and presents an approach that leverages existing manually created test cases to automatically infer MBT models that can identify additional ONBs that the manual tests missed. The second part presents a framework for the extraction and analysis of system invariants from automatically generated test cases using data mining. These automatically generated invariants yield useful insight into the actual system behavior and can reveal ONBs by identifying flaws and missing elements between the developer-maintained specifications and the implementation of a system.

The techniques presented in this thesis are intended to complement the approaches that address ONBs from the requirements-engineering and safety-engineering perspective by leveraging additional development artifacts that are available to testers, e.g. models, source code, and executables of the system, as well as execution data from test executions and log files.

1.1 On the nature of Off-Nominal Behaviors

While the term ONB is not standard software-engineering terminology, the notion is very important and deserving of more study. NASA recognizes the importance of ONBS and uses the concept throughout their engineering and software-engineering work [1, 13–15]. In standard software-engineering terminology some

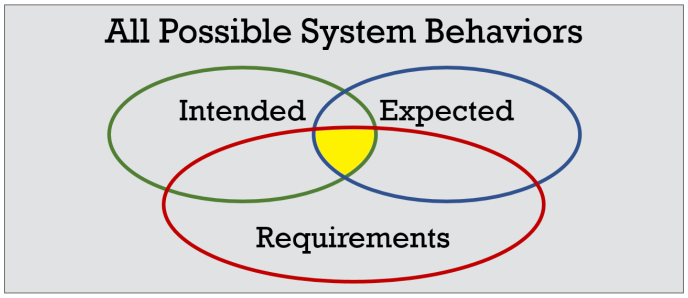
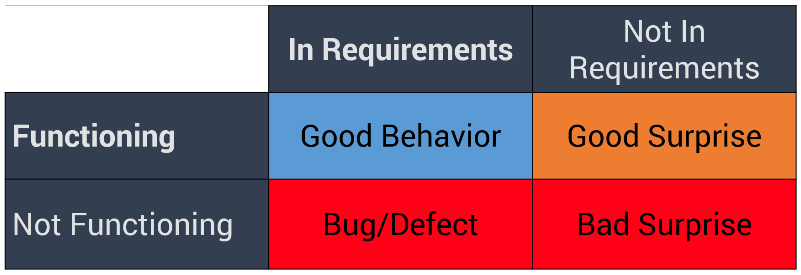


Figure 1.1: An illustration of the deviation between intended- and expected behavior and the requirements.

aspects of ONBs are covered under the umbrella of system robustness [16] and fault tolerance [17]. One of the goals of this work is to identify and document ONBs via software testing, so that the system can then be insulated from negative effects of these ONBs, making the system more robust and fault-tolerant.

In order to be precise about the notion of ONBs, this dissertation follows the definition of Day et al. [1], which states that ONBs, are behaviors of a system that are *"not intended or expected"*. Based on this definition, the nominal-behavior (NB) is then the intended and expected behavior of a system. The mentioned intentions refer to the intentions of the developers of the system, whereas the expectations refer to the expectations of the user/customer of the system. The intentions and expectations are codified in the documentation of the system. The documentation can be requirements, models, use cases etc.

Ideally, intentions, expectations and requirements should all coincide. However, in practice all three often diverge (see Figure 1.1) for a number of reasons (e.g. miscommunication or misunderstandings between the stakeholders and developers). The results are partly invalid and/or incomplete requirements. In addition to tradi-



	In Requirements	Not In Requirements
Functioning	Good Behavior	Good Surprise
Not Functioning	Bug/Defect	Bad Surprise

Figure 1.2: Classification schema for system behaviors.

tional issues with requirements (e.g. missing requirements, incomplete/inconsistent requirements [18]), a common problem is the focus on nominal behavior [2]. ONBs are often an afterthought, and the specification of their causes and how the system should react to the occurrence of them are therefore often lacking [1, 19]. This is also a problem in testing and oftentimes tests cover happy scenarios along an already beaten path [3] and do not cover ONBs. Mission- and safety-critical systems often operate in scenarios that are hard to anticipate and need to handle ONBs in a robust way.

The transformation of the documentation into the implemented system adds another layer of complexity to the problem. In addition to the already existing mismatches between intentions, expectations and the requirements documentation, this second transformation adds mismatches between the documentation and the implementation of the system. A source of such deviations are so-called implicit requirements [20], which can arise during development. They appear when programmers rely on their intuitions to make determinations about what ought to be required instead of what is actually mentioned in the requirements. Since it is hard to retroactively identify the intentions and expectations, this work treats all behaviors that are not documented as ONBs.

Based on the analysis of issues and missing requirements from our case studies, we define the following classification schema (Figure 1.2) for nominal and off-nominal system behaviors. There is one nominal behavior type (Good Behavior) and three different types of ONBs (Bug/Defect, Good Surprise and Bad Surprise). The term surprise is inspired by White et al. [21] who used it in the context of GUI testing to describe behaviors that depart from the expected behavior and are not explicitly mentioned in the specifications. This work extends this definition to two kinds of surprises: good surprises and bad surprises. The details of which are explained below.

- **Good Behavior**: *Good Behavior* encompasses all behaviors that work as documented, without any issues. All behaviors in this category can be traced back to a requirement or design artifact. This is the nominal behavior of a system. Testing often focuses on the documented behavior.

- **Bug/Defect**: *Bugs or defects* are traditionally issues in a systems that can be traced back to a requirement or design artifact. This is ONB: it is neither intended nor expected since it violates the requirements/design documents, which are the codification of the intentions and expectations.

- **Good Surprise**: The *good surprise* category contains behaviors that are functioning without detrimental effects to the system and serve a distinct purpose but are not documented in the requirements. This is ONB according to our extended definition, since it is undocumented behavior. This class of behaviors are often overlooked, and instead the focus is put on behaviors that tend to have detrimental effects on the system. However, undocumented good behavior can be problematic as well. System behavior that is not well documented can lead to back doors and therefore create security problems. Furthermore, undocumented behavior and can hamper the developers if the system has to be maintained or extended. Developers could break behaviors that the system's users rely on, since these *good surprises* are not documented.

- **Bad Surprise**: *Bad Surprises* encompasses all ONB with detrimental effects on the systems that cannot be traced back to a requirement or design artifact. A system crash due to an unspecified, and therefore missing, error-handling routine is an example of a Bad Surprise. Since such behaviors are not mentioned in the requirements or design artifacts these are hard to test for. The severity of the ONBs and the nature of the system determine if and how they shall be addressed. In less safety oriented systems (e.g. non-secure websites) *Bad Surprises* might be ignored unless the customers specifically complain about them. However, in safety-critical systems *Bad Surprises* can be dangerous. The developers should at least be aware that such an ONB exists in the system, how it is triggered and what its impact on the system is. This way developers can make informed decisions about addressing the ONB.

The objective of system testers and developers is to get all possible system behaviors into the Good Behavior quadrant. There should not be any undocumented or non-functioning behaviors in the system. Due to the complexity of modern soft-

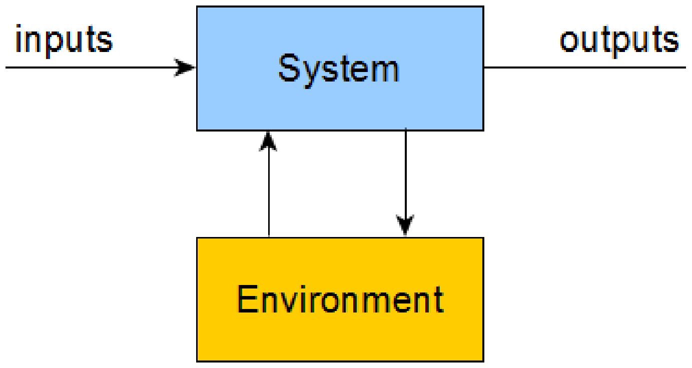


Figure 1.3: Schema of the input/output systems that we analyzed in this dissertation.

ware systems, this is however a nearly infeasible task. This work nevertheless aims to alleviate the problem by addressing all three categories of ONBs. The goal of this work is to identify and document the ONBs that a system exhibits, so that they can be addressed by the developers. In order address all ONBs bugs have to be removed and mechanism have to be introduced that handle inputs leading to bad surprises. By documenting the actual behavior of the good surprises they become good behavior and seize to be ONBs.

This work focuses on the subset of ONBs that can be caused by specific inputs to the system. The systems we are evaluating are input/output based systems (see Figure 1.3). The inputs to the system can have the form of sequences of input actions (e.g. connect to the system and then disconnect from it again). Each action of a sequence can have associated data (e.g. the id to connect to). This work does not address ONBs due to environmental factors that the system can be exposed to (E.g. physical connection to a system has been severed). This issue is left for future work.

1.2 Model-Based Testing to identify ONBs

In current practice, MBT techniques have been used to provide automated support for thorough evaluation of software behavior. MBT is a technique that employs models to describe the system under test (SUT) in an abstract manner and uses these models to derive test cases for that SUT [22]. There are many different MBT technologies and approaches that use different modeling notations. Neto et al. [23] surveyed and classified existing MBT approaches based on their modeling notation. State machines were the most common forms of modeling notations used in MBT, a finding that coincides with another survey of MBT practitioners [24]. State machine models are easy to comprehend, yet have a formal definition that can be leveraged for advanced verification techniques such as model checking.

A key contribution of this thesis is a technique [25] for generating state machine based MBT models on the basis of manual test cases, in order to better accommodate ONBs. This idea was inspired by three case studies of MBT [26–28] that we performed and that also represent a contribution to the thesis. Our case studies showed that the creation of the models and the associated test infrastructure to execute the resulting tests can be time consuming and needs skilled testers. During our case studies we also learned the value of existing manually created test cases of a system. They give invaluable insight into the usage and expected behavior of the system. In the case study of a web based system [26] we compared the effort and efficiency of the MBT approach against a manual tester in an industrial setting. Even though MBT found many serious issues the insights the manual tester had into the development process and the domain allowed her to find complex issues that MBT missed. In order to capture this valuable information we developed and evaluated a model and test infrastructure generator [25] that can automatically analyze existing tests for web-based systems and create testing models and their associated infrastructure from them. The models can then be used to create additional tests that cover hitherto untested and often off-nominal scenarios. In our evaluation we were able to identify additional ONBs in the system that the manual tests alone could not detect.

Our case studies evaluated the MBT approach in terms of its general effectiveness and efficiency by applying it to a variety of system types, e.g. web based systems [26], publish-subscribe style systems [27] and operating system wrappers [28]. Several additional issues were found, even in already well tested safety critical systems, and especially related to off-nominal inputs. Another benefit of the MBT workflow is that the formalization of the requirements as models is helpful with the identification of issues in the requirements [22]. The resulting models are well suited to convey knowledge from the testers to domain experts when they have inqueries about the system behaviors. In our studies the models were very useful to explain the detected ONBs to the developers and to discuss with them how the system should react to their occurrence.

1.3 Specification Extraction to identify and document ONBs

Accurate and complete information about the actual behavior, including its ONBs, are essential to developers and users of a software system. Developers need such information to make informed decisions during development, maintenance, evolution and testing; users need similar information to interact with the system properly and safely. Unfortunately, traditional manually maintained system specification documents are often incomplete, inaccurate, or out-of-date, and they can impede the development, testing as well as the effective and safe usage of the system after it has been deployed.

To counteract this problem of misleading specifications, researchers have proposed the use of *invariant mining* on run-time system artifacts [20,29,30]. Invariants describe properties and relationships that hold for all executions of a system. For example the invariant $\textit{speed} > \textit{25} \Rightarrow \textit{cruiseControl} = \textit{off}$, from a cruise control system, implies that whenever the speed is smaller than 25mph then the cruise control is not active. The intuition is that these invariants can convey information about system behavior at a much higher level than system code, while being free from the inaccuracies that often plague developer-maintained specifications. These techniques have been shown to yield useful information about systems, and even to reveal flaws in system specifications [20, 29, 30].

To document ONBs and to complete the specifications of a system with respect to its ONB, we created the Specstractor framework and tool chain which extracts invariants from automatically generated test cases using data mining. The

Specstractor framework is based on an approach by Ackerman et al. [20]. It learns invariants of a system that take the form of logical implications, where the left-hand side (LHS) of the implication represents inputs and internal variables of the system and the right-hand side (RHS) represent the outputs. For example the rule *brake = pressed* $\Rightarrow$ *active = false* corresponds to the natural language requirement *"The brake shall always deactivate the cruise control"*. The completion of the requirements with respect to the ONB is intended to help developers and testers to understand the system and how it behaves in off-nominal scenarios. This information can be used to design additional test cases or add routines to the system that mitigate the effects of the ONB.

Since data mining techniques rely on statistical measures to infer relationships between data, some of the invariants can be inaccurate. In order to address these false positives the framework employs an iterative $test \rightarrow infer \rightarrow instrument \rightarrow retest$ paradigm. Specifically, an initial set of invariants is inferred from the first set of automatically generated test cases. Additional testing cycles are used to identify false positives and to refine the proposed set of invariants until a final, stable set of invariants has been found. The automated test generator [31] can be given free rein to choose the input values and input sequences, including off-nominal inputs and it will explore ONBs as well as NBs of the system. Mining specifications from test cases that provide off-nominal input values and sequences will thus document these ONBs.

The underlying data-mining algorithms used by these tools typically are computationally complex, and this introduces efficiency problems when the techniques are applied to larger systems. In addition, as systems grow in size the likelihood of false positives also increases, thereby degrading the accuracy of the generated information. Our work shows how the problems of scalability and accuracy may be addressed using static analysis. We show that by adding simple data- and control-flow analyses into our invariant-generation technique we can improve both the efficiency of invariant mining and the accuracy of the invariants that are produced [32].

Our previous work on invariant mining [20,32] required variables to come from discrete types; they could not infer an invariant such as $\mathit{speed} > 25 \Rightarrow \mathit{cruiseControl} = \mathit{off}$ without the help of manual abstractions supplied by the user. To provide better support for continuous variables we have developed the Range Miner algorithm, which uses a combination of automated data binning, association-rule mining and consecutive merging and adjustment of the invariants to identify accurate value ranges.

In order to guide the user through the generated data, Specstractor contains the web-based *Insight Tool* for the analysis of the resulting invariants. It allows the user to navigate (search, filter) the resulting invariants and creates high level views of the invariants in the form of heat maps and state machines.

We evaluate the Specstractor tool chain by applying it to systems from the automotive and medical-device domain and compare the extracted invariants against existing requirements specifications for these systems. We show how the invariants may be used to identify issues in the requirements documentation and how deviations between the requirements and the system and inconsistencies in the requirements manifest in the invariants. We further show in which cases the approach is unable to identify requirements accurately and why it is unable to do so. Finally, we present effort data to demonstrate how much time it takes a tester to apply the approach to compare the resulting invariants against real world specifications artifacts.

1.4 Research Hypotheses

The research for this proposal is motivated and guided by the following three research hypothesis:

1. MBT can be used to identify and test ONBs in software systems.

2. Manually created test artifacts can be used to automatically infer testing models and their associated test infrastructure. The test cases generated from these models can identify ONBs that the original tests missed.

3. Through Test Generation and Data Mining Techniques characterizations for ONBs can be identified and used to improve the requirements.

The first hypothesis is concerned with the identification and testing of ONBs using MBT. This thesis evaluated the usage of MBT in terms of its effectiveness (E.g. can it identify ONBs in a system) and efficiency (how much effort does it take to apply it) in three case studies.

The second hypothesis is concerned with the improvement of the MBT process, using existing manually created test artifacts. As part of the evaluations of MBT we learned that existing manually created test cases can be very valuable for the MBT process. We developed an approach that leverages existing manually created test cases to infer testing models that can then produce additional tests that more effectively identify ONBs.

The last hypothesis is concerned with the completion of the requirements with regards to ONBs. The argument is that through a combination of test-generation and data-mining techniques, the true behavior of a system can be learned and the information gained can be used to improve the requirements with respect to its ONBs.

1.5 Summary of Contributions and Thesis Outline

The contributions of this thesis are:

- Two empirical case studies evaluating MBT. The studies evaluate the effort and efficiency of the approach to identify ONBs in already tested NASA systems. (Chapter 3)
- One empirical case study comparing MBT to Manual Testing for testing of a web based system. (Chapter 3)
- An approach to automatically generate testing models and test infrastructure from existing test cases of a web based system. The generated test cases from these testing models can identify ONBs that the original manual tests missed. (Chapter 4)
- Development and Implementation of Specstrator, a framework and tool-chain for the extraction and analysis of system invariants from automatically gener-

ated test cases. The resulting invariants can be used to identify and document ONBs of a system by comparing them against the existing specification artifacts of the system. (Chapter 5)

- Introduction of static analysis in the form of control- and data flow analysis into the Specstractor framework to improve its effectiveness and efficiency. (Chapter 6)

- A new algorithm that leverages the iterative cycle in Specstractor to infer invariants involving continuously-valued variables with few false positives. (Chapter 7)

- An evaluation of the Specstractor tool chain by applying it to systems from the automotive and medical device domain. The resulting invariants are compared against existing requirements specifications of the systems. The evaluation shows what type of ONBs can be identified and how they manifest in the resulting invariants. (Chapter 8)

The rest of the thesis is structured as follows. Chapter 2 contains background information about some of the technologies and terminologies used in this dissertation and presents the related work. Chapter 3 presents the three MBT case studies, while chapter 4 discusses the Model Generator that uses existing tests to create MBT models. Chapter 5 describes the Specstractor framework for the extraction and analysis of system invariants and discusses the shortcomings that we addressed as part of this dissertation. Chapter 6 introduces and evaluates the addition of information from static analysis to improve the accuracy and performance of the approach. Chapter 7 present the Range Miner algorithm, which can infer invariants over continuously-valued variables. Chapter 7 evaluates Specstrator by applying it to systems of the automotive and medical device domain and comparing the resulting invariants against actual requirements specifications in the form of state machines and natural language requirements. The last chapter concludes this thesis and discusses future work.

+

Chapter 2: Background and Related Work

This work contains two parts that both address the problem of ONB from the testing perspective. In the first part we perform evaluations of MBT to test for ONB and developed ways to improve the MBT process using existing manually created test cases. In the second part we are employing invariant mining to automatically extract specifications of a system using automated test generation and data mining. This chapter provides background information for the reader about the MBT process and about the technologies that we leverage in our invariant mining approach. Following the background information we provide an overview about the relevant related work.

2.1 Model-Based Testing

This section describes the general work flow for MBT that we followed in our evaluations of MBT that are presented in chapter 3. MBY employs the ideas of Model-based design and applies them to software testing. Figure 2.1 is used to illustrate the approach. MBT employs models to describe the system under test (SUT) in an abstract manner. The model is used to derive test cases that are then executed against the implementation of the system. There are different MBT technologies.

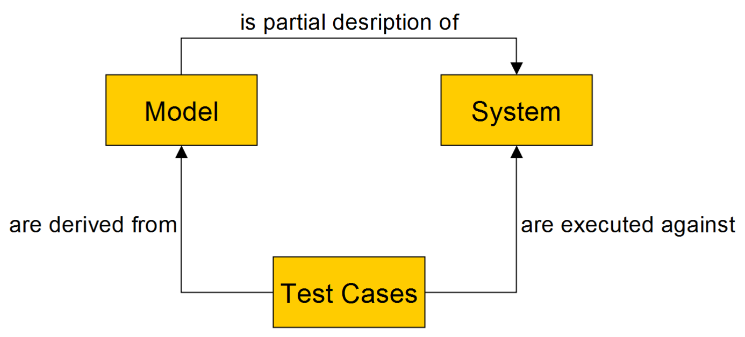


Figure 2.1: Illustration of the Model-Based Testing approach.

This section explains the approach that is used in this work, which is based on a description of state-machine based MBT in [22]. A simplified example model that describes part of NASAs Goddard Mission Service Evolution Center (GMSEC) system from the case study in chapter 3.2 is used to illustrate the approach. GMSEC is a flexible and reusable framework that provides a publish-subscribe style communication framework between applications via a 3rd party message bus. In order to communicate an application creates a connection and then connects to the bus. It can then send messages or subscribe to topics and receive messages that are sent by other applications or by itself. The messages are routed via the message bus into the message queues of the subcribers.

The modeling notation in this work are finite state machines (FSM) and extended finite state machine (EFSM) models, which represent the expected behavior of the SUT. The model in Figure 2.2 describes how to *create* connections, *connect* them to the bus, *disconnect* them from the bus, and *destroy* them. Models consist of 1) *transitions* (arrows in the model), which represent requests (or stimuli) to the SUT, and 2) *states* (boxes in the model), which represent the conceptual state that

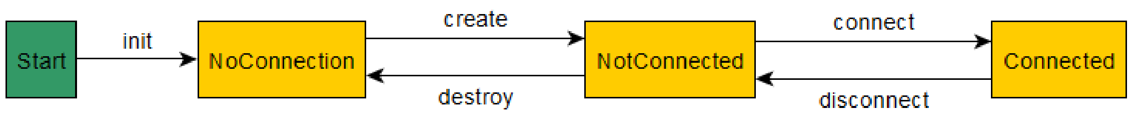


Figure 2.2: FSM testing model for the GMSEC system. It describes how a system can connect to and disconnect from the message bus

the SUT is expected to be in after the requests have been executed. In the example model, the transition *create* relates to an action that creates a connection object in the SUT and the state *NotConnected* indicates that the SUT should contain a connection object but not be connected to the bus yet.

The MBT workflow used in this work is visualized in Figure 2.4. The tester first builds an understanding of the SUT by reading the documentation (e.g. requirements, use cases, existing test cases), by using the SUT and by talking to the stakeholders of the SUT. In our experience, analyzing existing test cases of the system was particularly valuable. But instead of writing test cases directly, like a manual tester, the MBT tester models the functionality of the SUT and the expected behavior using a model, encoding the SUTs expected behavior. After the model is finished, the tester employs a tool that derives test cases from the model. A test case is a traversal through the model from the start state to the point where the stopping criterion is met. The tester can use a variety of traversal algorithms (random, shortest path) and stopping criteria (transitions coverage, reaching a certain state) to create test cases. During the traversal, the tool stores the path (the list of transitions and states that it visited). These paths are called abstract test cases.

Table 2.1: The model element create is mapped to its implementation in Java

Model Element	**Implementation**

Create	status = ConnectionFactory.create(con);
	
	assertTrue(status == GMSEC_SUCCESS)

By mapping the states and transition of abstract test cases to executable actions, these abstract descriptions are translated into concrete (automatically executable) test programs that can test the SUT without assistance of the tester. This is done by mapping each abstract action and reaction to code fragments. We call these mappings the test infrastructure. An example is provided in Table 2.1 where the transition *create* is mapped to the Java code that creates a connection and then checks whether the return value was as expected. The generated test cases are then executed against the system and the results analyzed to identify the issues of the system.

EFSMs extend FSMs with the concept of variables and guards. The example in Figure 2.3 uses the *msg* variable (short for message) to keep track of the number of messages that have been received by the system. The value of the variable is increased whenever a new message is published (*publish/msg++*) and decreased when a message is taken out of the message queue (*getMsg/msg–*). Guards (*[msg == 1]*) are Boolean expressions that can be used to make advanced navigational choices in the test generation process. If a guard evaluates to true, its corresponding transition can be traversed. If it evaluates to false, the transition cannot be traversed and another available transition has to be taken. EFSMs can describe more complex

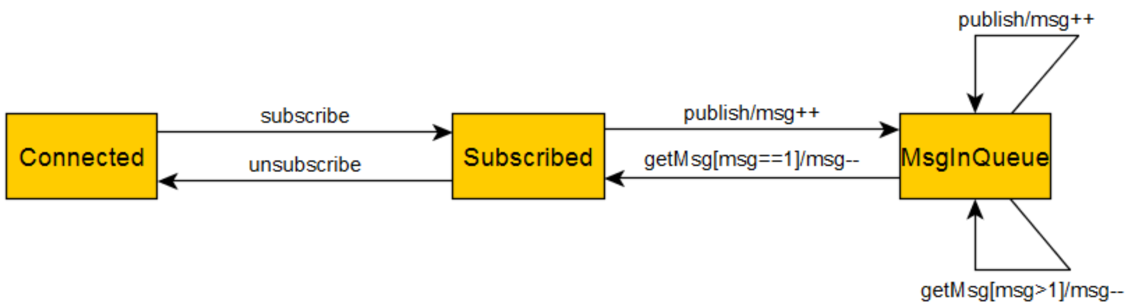


Figure 2.3: A EFSM testing model of the subscription and publishing functionality of GMSEC. The action publish employs data abstractions, since the name and contents of the message are not considered and it also uses communication abstraction. Furthermore publish/get make use of a temporal abstraction, since they do not consider the order of the publishing of messages.

scenarios than FSM. However, the testers that create EFSM models should have at least basic programming skills. Also, analysis and debugging of an EFSM model is more complex than for an FSM model.

Since it is infeasible to test all possible combinations of actions and also combinations of data for these actions we need to apply abstractions. Abstractions are simplifications to approximate the system under test, in order to reduce the complexity of the model. A testing model by definition is already an abstraction of the SUT. This work follows the definition established in [33] that describes abstractions for MBT. They refer to four classes of abstractions for MBT.

- **Functional Abstractions** omit details that the modeler deems are not necessary for the verification process. This focuses the model on the main functionality to be tested. It also allows for different models that describe different

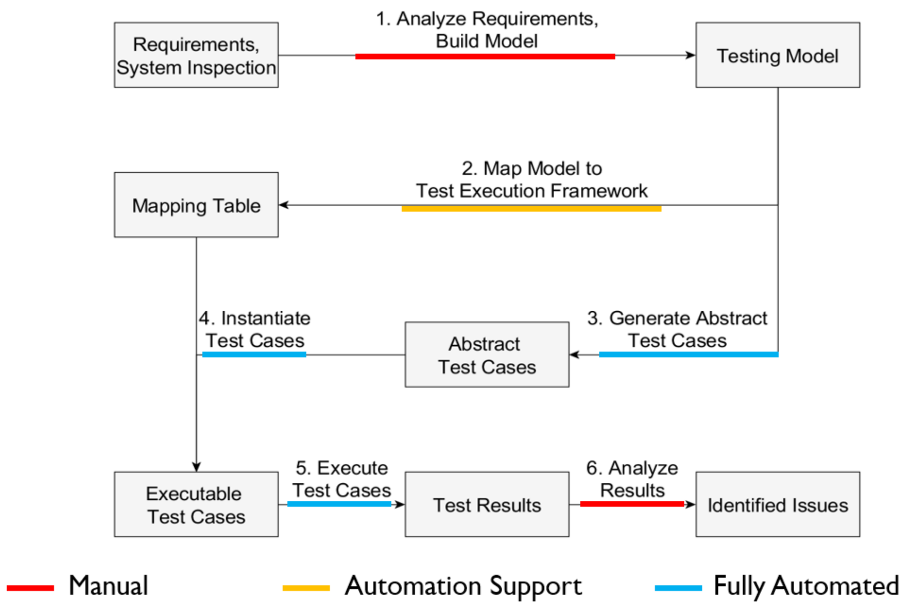


Figure 2.4: FSM testing model for the GMSEC system. It describes how a system can connect to and disconnect from the message bus

aspects of the system independently of each other.

- **Data Abstractions** map concrete data types in the system to abstract data types in the model. This helps to reduce the data complexity. For example instead of using the full range of possible input variables for a system the model only uses equivalence classes of the input.

- **Communication Abstractions** group together several interactions of the SUT into one abstracted action in the model. The abstracted action is then treated atomically in the model. This fixes the order of the events, and different interleaving are not possible anymore, even though they might be possible on a concrete level. An example of such an abstraction is the aggregation of handshaking interactions into one action. For test-case instantiation, these abstract operations can simply be substituted by the corresponding interaction. For building models, communication abstractions are often combined with functional abstractions. The actions still need to be tested but this can be done in seperate models or unit tests.

- **Temporal Abstractions** are concerned with the simplification of real time and the ordering of events. For example one temporal abstraction would be to ignore the time of events and just focus on the order, or even ignore the order and just focus on the reception of the events. Another example would be a timer, which is abstracted in the model with two symbolic events. One for the start of the timer and one for indicating the expiration of the timer.

The model in Figure 2.2 is a functional abstraction of the GMSEC system, since model only focus on the connection part of GMSEC and do not consider subscription and publishing actions. The model in Figure 2.3 is an extension of the model in Figure 2.2. It adds the message subscription, publishing and receiving part. The model is an extended finite state machine model that uses the variable *msg* to keep track of the number of messages in the system. This helps keeping the state space of the model smaller than in a corresponding finite state machine. In order to model an equivalent as a FSM one would have to create one state per message. In this example, the state space of the FSM would even be infinite, since it can theoretically create any number of messages.

The model employs data abstractions since the data parameters (name, contents) of the messages that are published are not considered. It also contains communication abstractions in the form of the publishing action. In the real system, publishing consists of two parts, creation of the message and consequently publishing it to the bus. The atomic action publish in our example model groups the two actions together into one. Furthermore, *publish* and *getMessage* actions make use of temporal abstraction, since the model and test infrastructure does not consider the order in which messages appear in the queue and are only testing if the messages have been placed into the message queue at all. As can be seen in the examples, the models combines several abstractions together, even within a single transition.

The abstractions have to be replaced through the mapping in order to create concrete executable test cases. For example if a transition in a model groups together a sequence of two concrete actions in the system, then the mapping needs to include the two system callse that represent the action. Depending on the type and thoroughness of the abstraction, these mappings vary in complexity. Also some of the abstractions can be completely reversed or only partially reversed. For example, the model in Figure 2.3 does not consider the order in which messages are received and only checks if they have been received at all. In the mapping, one could introduce a list that keeps track of all the messages that have been published and checks that they have been received in the order in which they have been published. With the help of this list the abstraction could be reversed completely, without it it would be only a partial reversal. So the complexity in model-based testing can be divided and distributed between the model and its mapping.

2.2 Simulink

Simulink is a graphical block-diagram modeling language developed by The MathWorks Inc. Simulink is used in conjunction with The MathWorks' MATLAB® tool[1]. An example of a Simulink model can be seen in Figure 2.5. The example is a model of a cruise control system that determines whether the system should be active or not, and what the current throttle for the engine of the car should be. Blocks represent functions that perform a calculation based on input data and produce an output. Data is propagated between blocks via lines that connect the outputs of one block and go to the input of another block. Simulink models may also contain hierarchical state machines in the Stateflow® hierarchical notation (the yellow block in the example).

Because of its ability to model both discrete- and continuous-time dynamics, Simulink is widely used in the automotive, ground transportation and aerospace industries to design control systems. Discrete Simulink models are also often used as specifications for embedded control software, and tools such as The MathWorks' Embedded Coder and dSpace's TargetLink can automatically generate deployable C code from these models.

[1]MATLAB® and Stateflow® are registered trademarks of The MathWorks, Inc.

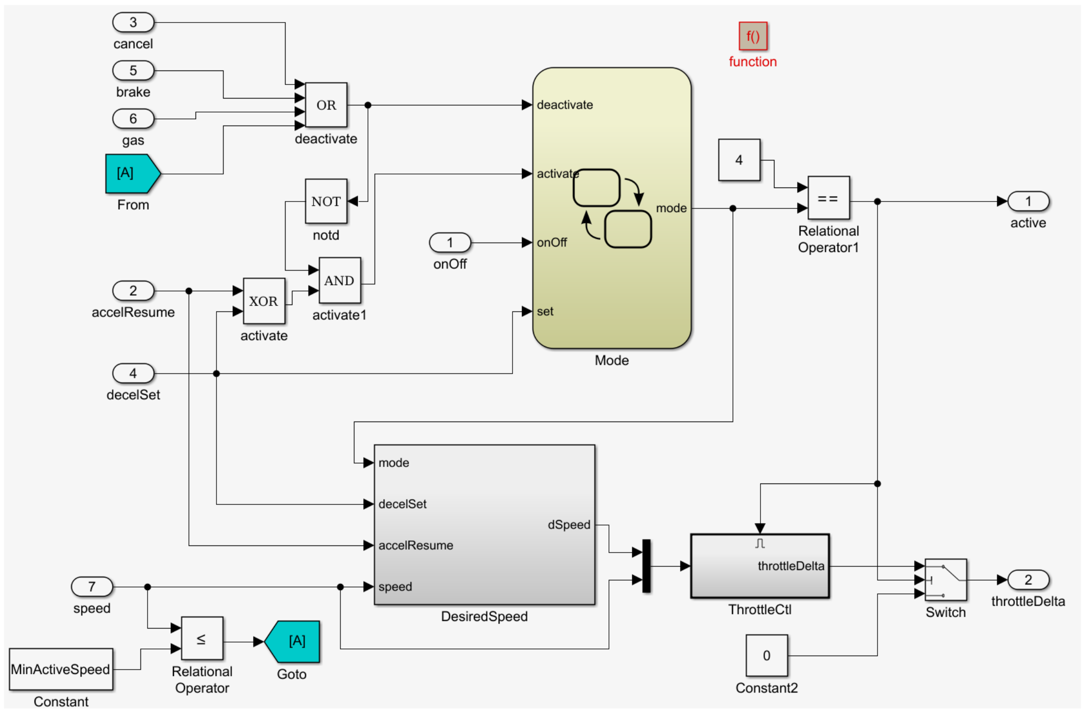


Figure 2.5: Simulink model of a cruise control system. The yellow block with rounded corner contains a state-machine in the Stateflow language.

2.3 Reactis

Reactis[2] [31] is a model-based testing tool that automatically generates test cases in the form of sequences of input vectors for models in the MathWorks Simulink / Stateflow notation. The goal of Reactis is to create test data that actively attempts to cover structural coverage targets of a model (e.g. decision coverage, subsystem coverage, state/transition coverage). Generally, for reasons of undecidability, full coverage cannot be guaranteed. Reactis uses a combination of random test generation, heuristics and symbolic execution in a two phase approach with the goal of maximizing the coverage of the test cases that it creates.

In the first phase it generates a set of semi-random inputs where the user can specify the number of tests and their length. Heuristics and static analysis are used in the first phase to identify interesting input values for the system (e.g. always test 0, the boundaries of the value range, or use the value of constants identified via static analysis). These are then added in addition to the ones chosen by the heuristics. In the second phase it uses reverse symbolic execution to try to cover the targets that are not covered by the random tests yet. The details of the technique are covered by US Patent #7,644,398.

2.4 Instrumentation-Based Verification

This is a summary of the Instrumentation-Based Verification (IBV) approach presented in [20, 34]. Models are the central element in Model-Based Development

[2]Reactis® is a registered trademark of Reactive Systems, Inc.

(MBD) and because of their centrality it is important that they behave correctly and conform with the requirements of the system. Reactis supports a verification technique for design models called Instrumentation-Based Verification (IBV) [34]. In IBV each requirement is modeled as a monitor model. Monitor models take the form of a Simulink model or textual assertion. Reactis is then used to instrument the model that is to be verified with the monitor models. The user then generates test cases with the goal of covering the instrumented model, including the constructs contained in the monitor models. While the tests are being generated, Reactis searches for counter examples to the monitor models that would invalidate the monitor models. Tools such as The MathWorks Design Verifier®[3]® employ model-checking for to verify model correctness. The advantage of IBV in contrast to model checking techniques is that as a testing-based approach it scales better but that comes at the price of the iron-clad guarantees of correctness that model checkers can give, when they do indeed terminate.

2.5 Association Rule Mining via FP-Growth

In order to mine association rules from test data, we employ the FP-Growth algorithm [35]. The algorithm works in two steps.

In the first step it identifies frequently co-occurring sets of *items*. Co-occurence in the context of our work means that the items appear together in the same test step of a test case. Consider {*in1=1, in2=1, out1=1*}, this is a set of 3 items or

[3]Design Verifier

3-item set. Each item set has a support value that indicates how often the items in the set appear together in the test data. For example if the item set has a support of 5 there would be five co-occurrences of in1=1, in1=1 and out1=1 in the test data. In order to identify this item set the algorithm first identifies all frequent single items by counting the support (number of occurrences) of each item in the test data. In order for an item to be frequent it has to have more support in the test data than the support threshold, which is a user defined value.) in the test data. If it appears frequently enough the item is retained, otherwise it is eliminated. After identifying all single items hat appear frequently enough the algorithm then identifies the frequently occurring 2-item sets, 3-item sets to n-item sets (n in our case is bounded by the number of inputs, internal variables and outputs). For information on how the FP-growth algorithm identifies these item-sets efficiently we refer the reader to [35]. The output of this step is a list of all single items and itemsets in the test data that are above the user supplied support threshold.

In the second step the algorithm creates association rules by taking one item from an itemset to be the RHS of the rule, with the remaining items in the set forming the LHS of the rule. For example for the itemset {*in1=1, in2=1, out1=1*} it would create the three rules 1.) *in1=1* $\wedge$ *in2=1* $\Rightarrow$ *out1=1*, 2.) *in1=1* $\wedge$ *out1=1* $\Rightarrow$ *in2=1* and 3.) *in2=1* $\wedge$ *out1=1* $\Rightarrow$ *in1=1*. The algorithm then calculates how often the rule is true, which is the so called confidence of the rule. If the confidence passes a given threshold the rule is accepted. In our case the threshold is 1.0, which means there are no counterexamples and the rule is always true.

Association rules take the form of implications where the LHS implies the

RHS. In the case of our work we are only interested in rules where the LHS of the rule only contains inputs and internal variables of the system and the RHS contains only outputs. We therefore modified the algorithm so that only the first of the three rule is produced since an input value appears on the RHS in the other two rules.

2.6 Related Work

This section discusses the related work of the dissertation. It is grouped into 1. Off-Nominal Behaviors; 2. Model-Based Testing and 3. Specification Extraction.

2.6.1 Off-Nominal Behaviors

Nancy Leveson analyzed several failures in safety critical systems [3,5,36] and identified lack of testing ONBs as one of the main causes for the failures in several cases. Furthermore, according to Leveson software requirements often only specify what the system shall do and not what the system shall not do or how the system shall behave in cases of an unknown event. This can be seen by the fact that ONBs are actually not a common term in software engineering. However, there are some approaches that address ONBs from the requirements engineering perspective [6–9] and some mention it in the context of software testing. For the testing perspective, these are often related to robustness, security and reliability testing [13,37–40].

2.6.1.1 In Requirements Engineering

There are some approaches that address ONBs from the requirements engineering perspective [6–9]. Not all of them necessarily use the term ONBs but the ideas behind the following papers are similar to the goal of this work which is the identification of potential causes of ONBs and their impact on a software system. Alexander [7] introduces the idea of misuse cases, which describe how the system can be misused and how the system shall react to the misuse. Ishimatsu and Leveson [8] present a hazard analysis technique that can identify hazardous design flaws, which take off-nominal commands into account. Aceituna [9] introduces an approach that formalizes the requirements of a system as a set of concurrent state machine models and then employs model checking to identify missing, or invalid behaviors. Day et al. [6] seek to identify causes for failures and their potential propagation through the system by modeling the system and its ONBs in the SysML modeling language.

The assumption of our work is that it is hard or nearly impossible to identify all ONBs without the actual implementation, or a prototype of the implementation, since there are so many potential causes for off-nominal behaviors [12]. Our work proposes a complementary approach to the approaches that address ONBs from the requirements engineering perspective. Our approach leverages the artifacts from the testing process (e.g. models, source code, executables of the system, as well as execution data from test case executions or log files). ONBs are identified with the help of Model-Based Testing and Specification Extraction and the resulting artifacts are used to document the ONBs.

2.6.1.2 In Software Testing

Robustness testing covers some aspects of ONBs, specifically ONBs in the Bad Surprise category. IEEE [41] defines robustness of a system as ”the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions”. Fernandez et al. [37] present a model based approach for robustness testing that leverages fault models and scenarios for test case generation. Robustness can be addressed with model-based testing technologies by generating many tests or tests with unknown scenarios. Our specification extraction approach also considers unexpected positive scenarios which robustness testing does not.

Fuzz testing [38] is another type of robustness testing that is mostly aimed at detecting security-relevant weaknesses of a system. In fuzz testing invalid, unexpected, or random data is sent as inputs to a system to identify potential issues in the system (e.g. crashes, memory leaks) that can be exploited to breach the security of a system. The Reactis test generator that we employ for specification extraction combines random data generation similar to fuzz testing with symbolic execution techniques. Additionally, we employ fuzz testing to generate new tests based on the tests created by Reactis and the identified invariants that help to remove false positives that Reactis may miss.

Ghosh et al. present two approaches [39] [40] that use fault injections to test off-nominal scenarios of software systems, thereby improving test generation with regards to the occurrence of ONBs. Both approaches use tools that instruments the system under test and can inject faults during runtime. While Gosh's approach tests issues caused due to environmental conditions, our work focuses on the detection of ONB via the input and output interface of the system.

Foyle [13] introduces an approach that is intended to improve system design through the use of off-nominal testing. While the descriptions sounds very similar to the approach presented in this work, the application area is different. Their work is targeted at identifying training issues and procedures for human in the loop testing.

2.6.2 Model-Based Testing

2.6.2.1 Evaluations of Model-Based Testing

Several modeling notations for MBT exist [23,24]. We decided to use graphical state machine based notations since they have a short learning curve and based on our experience and a survey of MBT users [24] are more appealing to practitioners. Other state machine based MBT approaches use declarative languages for modeling. In Microsoft's SpecExplorer tool [42], models are specified as C# programs and an FSM is then generated that contains all the behaviors that the program can execute. The resulting state machines can be quite large (several thousand states and transitions). But from our experience smaller state machines are very valuable to convey information about the system behavior, which was very useful in our case studies when we had to correspond with developers of the system.

MBT approaches have been evaluated mostly in terms of their effectiveness [43] [44], where they show MBTs' promising potential. The OSAL and GMSEC case studies also fall in these categories of papers that mostly looked at the effectiveness. Reza et al. [45], Heinecke et al. [46] and Andrews et al. [47] presented state based MBT approaches to test web based systems. They showed that it is feasible to use these techniques to test web based systems but no comparison to other approaches and no effort or issue data was presented.

However, in order to persuade potential adopters of the usefulness of MBT, more studies should focus on the efficiency of the approaches in terms of the effort that is necessary to apply them. A notable exception is Grieskamp et al. [48] who studied the effort involved in manually coding test programs in contrast to test programs that were generated by MBT. Our MBT vs Manual study (see chapter 3.3), was, as far as we know, the first designed empirical study that compares cost (effort) as well as benefits (detected defects) related to completely manual testing (no automation whatsoever) to MBT where the two approaches tested the same two versions of the SUT in parallel.

A study that evaluates MBT and captures replay techniques for performance testing was published, after our evaluation of MBT vs Manual, by Rodrigues et al. [49] Even so the testing goal was different from the MBT vs Manual case study (functional testing vs performance testing) their findings are similar. For smaller testing tasks without regression testing, the manual approach requires less effort. For larger tasks and if the test cases have to be changed in order to test a new version, MTB becomes more valuable.

Apfelbaum and Doyle [44] conducted a case study of a long distance call service and a work order system. They modeled the system as FSMs and compared this approach with manually created test cases. The focus was mainly on the modeling process and not on the analysis of the effort or the detected issues and comparison to the manual approach. Utting and Legeard [22] compared effort of Manual Testing, Capture Replay Testing, Testing via Scripts, Keyword Testing and MBT. Their study was done with a theoretical model of the different approaches and they calculated the effort it would take to produce test cases with the given technique for a hypothesized system. In contrast, the Elexnet study collected real world test data and compared the effort and effectiveness of manual testing and MBT based on testing of a SUT that was professionally developed.

There are numerous papers devoted to code coverage primarily for measuring the (in-) adequacy of tests as well as to demonstrate strengths of test generation algorithms. In the context of MBT, at the model level, there are metrics such as number/percentage of states and transitions covered as well as the length of the generated tests, etc. We surprisingly did not find many papers that discuss both code and model coverage. In the OSAL case study (see chapter 3.1), we used code coverage to manually identify behaviors that were missed in the model, and to update the model to cover missing behaviors for achieving better code coverage. In this regard, the work by Kicillof et al. [50] is related to our work. They combined model-based testing for test procedure generation with symbolic execution (at code level) for data parameter generation in order to achieve code coverage for .NET applications.

2.6.2.2 Model Generation to support Model-Based Testing

There are three main types of techniques that are frequently used for state machine model inference. The first type of techniques are based on Angluins L*algorithm [51]. These approaches use a learner that knows all possible input and output actions of the system and a teacher that can for a given sequence of input actions provide the correct output action. In our context the teacher would be the SUT and the learner would create model hypotheses and compare them to the SUT until it has found a model that corresponds to the behaviors of the SUT. Tretmans [52] discusses how learning algorithms can be applied in the context of MBT. In our approach we do not create a model of the system by hypothesizing and learning but generate a model from a set of manually created test cases using heuristic rules that are based on the structure of the SUT itself.

Another common type of techniques are based on the k-tail algorithm [53], or variants of the k-tail algorithm [54]. The k-tail algorithm creates candidate models by observing execution traces and employing heuristics to merge different executions. In k-tail algorithms states are defined by the future behaviors that can occur from them. In our model generation technique states are based on unique identifiers that are based on the assert commands of the test data. Although k-tail based algorithms can be applied on test case sequences in our experience there were often not enough manual test cases to create an accurate model. These techniques often rely on data that contains more sample behaviors, like system logs for example.

Lastly, another common approach to generate models are based on observations of the runtime state of the system, or an abstract representation thereof. Marchetto et al. [55] present an approach that abstracts the DOM model of a web application into a state model and then generates additional test cases. The states in the presented model generation technique have only a small relation to the underlying SUT and the model is not as understandable. Our model on the other hand follows the flow of the website and as part of the unique identifiers for states some Selense assert commands have references to objects in the SUT (e.g. ids of buttons, text on the page). They applied their technique on an open source to-do list manager application with seeded defects. We applied our approach on an industrial web based system.

Memon et al. developed the GUITAR tool [56], which dynamically reverse engineers an event-flow [57] model of a system by automatically crawling its GUI. The obtained model is a representation of all the possible actions the user can do in that GUI system. Furthermore, they developed DART [58] which uses the extracted models for regression and smoke testing in daily/nightly builds. A challenge with the approach is supplying it with inputs for the GUI (e.g. like text for a text field). The approach used in this case study leverages the existing inputs from the manually created test cases and embeds them in the model.

The discussed techniques are very powerful and can test many behaviors of a system. However, oracles have to be added manually after the fact, which is not always trivial due to the size of the models. These techniques are therefore often used for regression testing and smoke testing. Our approach leverages the oracles from the original test cases and encodes them into the generated model for functional testing automatically.

2.6.3 Specification Extraction

2.6.3.1 Invariant Mining

Program invariants have ample uses in software development and software testing. Program invariants have been used to reduce the search space of model checkers [59], for the analysis of log files [29], for predicting incompatibilities between components [60], and to support program verification [61, 62].

This work employs invariant mining to extracts specifications of a system. The invariant mining approach in this dissertation is an extension to the work done in [20]. As part of this dissertation we evaluated the approach and improved it with regards to shortcomings that we identified. We added information from static data- and control-flow analysis to improve the accuracy of the resulting invariants and the performance of the overall approach so that we could analyze more complex models (see Chapter 6 and [32]). Furthermore, we added a new mining algorithm that can infer accurate invariants from continuously valued variables (see Chapter 7.1. The Specstractor framework has been applied to several systems of the automotive and medical device domain in order to evaluate its performance and to show how the resulting invariants compare to actual requirements artifacts and how much effort it takes to compare the resulting invariants against the requirements.

One of the earliest techniques to generate invariants is based on abstract interpretations [63, 64]. In this technique approximate symbolic execution is performed until an assertion is reached that any further executions of the program are unable to change. The assertion is hence an invariant. Due to scalability issues of the technique, other forms of invariant generation approaches have since been proposed.

InvGen [62] is an invariant generator for imperative programs. It uses a constraint-based approach to create invariants and uses static and dynamic analysis to support the constraint solving. The input to InvGen is a set of transition relations that encodes the behavior of the program. Programs written in a subset of the C programming language can be translated into this input notation with the help of an additional tool. In contrast Specstractor infers invariants from test cases via data-mining. Specstractor can be applied to any system as long as an automated test generator for it exists.

The Daikon system proposed by Ernst et al. performs dynamic detection of "likely invariants" in programs written in C, C++, C#, Eiffel, F#, Java, Perl, and Visual Basic. The invariants in Daikon are used as assert statements, documentation and formal specifications. Input data is obtained by executing existing test cases on an instrumented version of the system, but the data collected in this way does not guarantee good coverage of the evaluated program. By using monitor models to validate the inferred invariants, Specstractor can provide stronger guarantees for the correctness of the invariant set. Both Daikon and InvGen offer several templates for different types of invariants. For this thesis we focus on the extraction of implications since they relate well to textual requirements. For future work we are planing to evaluate other types of invariant templates to identify other useful invariants that help with the extraction of system specifications.

The ICE framework [61] present a learning based approach, based on templates, to infer invariants. Like Specstractor it also leverages an iterative approach to refine an initial set of invariants. The goal of ICE is to infer invariants that can be used in program verification and the invariants are very close to the code of the system. For Specstractor we focus on implications that relate well to natural language requirements of a system. Furthermore, these implications can entail all possible inputs, internal variables and outputs of the system and become very complex to the point where a search based approach with many templates does not scale well.

A common technique to extract system behaviors is to extract state-machines models, which are supposed to represent the system specifications [54, 65–68]. Variations of the three model inference techniques (L* [51], K-Tail [53] and system state observation [68]) are commonly applied to extract state machines from system behaviors. Sequence Diagrams are another common way of representing extracted behaviors from system executions [69–71]. In contrast, our work is concentrated on extracting individual requirements rather than creating models of the overall system behavior. Our intuition is that the single invariants are easier to compare against natural language requirements than complex state machines. However, the resulting state machine models of the above mentioned approaches could be leveraged as alternative representations of the systems behavior or as inputs for temporal logic query checking.

Some of these state-machine based approaches [65, 68] use the inferred models to create additional test cases for the system and use them to observe new system behaviors. This is similar to our work, which leverages the inferred invariants or hypothesis thereof to generate additional test cases to observe new behaviors and to verify the existing invariants. However, in our work the test generator is somewhat independent of the invariants and can target all system behaviors and by doing so is trying to identify counter examples to the invariants. The tests in the above mentioned work are limited to what the inferred models can describe. Beschastnikh et al. [72] present an approach that leverages inferred invariants for the refinement of extracted state-machine models.

Other specification-extraction approaches, such as the Refinement Calculus of Reactive Systems (RCRS) [73], creates contracts of a Simulink model in a bottom-up modular fashion, producing a specification that describes the global behavior of the model. Rather than specifying globally what all the behaviors are, our approach creates several invariants that are partial descriptions of this global behavior that can be understood independently.

Similar to our idea to use invariants to detect ONBs, specification mining approaches have also been used for the detection of incorrect error handling [65, 74]. Pradel et. al. [65] mine state-machine specifications for bug detection in Java programs. The resulting state machine is annotated with unhandled exceptions that are detected during the execution of a program. The CAR miner [74] uses sequence association rules in order to document exception handling behavior of a program. Both approaches instrument the system in a similar fashion than Daikon and the resulting specification contain code constructs. Specstractor does not collect such detailed information from the code and focuses on the inputs, internal variables and outputs of the system. While the test cases that Reactis produces will trigger exceptions and crashes, we do not have a notion for exception and crashes in the resulting invariants that Specstractor mines yet.

Most of the above mentioned approaches evaluate how many issues they found on a series of systems as we do with Specstractor. But for Specstractor we also collected effort data of the user. We show how long it takes to compare the resulting invariants against the specification artifacts. This also includes the time to debug the system in case that there are deviations between the invariants and the requirements.

2.6.3.2 Static Analysis

Translating Simulink models to other modeling notations and languages has often been done for the purpose of formal verification [75, 76]. Pantelic *et al.* [77] present a tool suite that can extract interfaces from Simulink submodels using static data/control-flow analysis. Instead of extracting interfaces our approach extracts invariants of the whole system. The Artshop tool [78] is a suite of tools for analysis, reporting and inspection of Simulink models. It has the capability to do static dependency/control-flow analysis, which is used to create slices of models. However, it does not have a specification extraction tool.

Lublinerman et al. [79–81] introduce several methods for dependency analysis of Simulink models that they use for modular code generation. Our graph representation and transformation process is similar to theirs *block-based dependency analysis*, but we are not employing it to give semantics to a diagram and are only interested in computing the dependencies between elements in a model. Their work also presents a *input-output dependency analysis* based on bipartite graphs. However, this is not applicable in our context, since we also need information about internal variables of a system.

The GoldMine tool [82] extracts temporal and propositional invariants for hardware designs using a decision-tree algorithm. It uses static analysis of the hardware design and formal verification to remove spurious invariants produced by the data miner. In our approach we created a static analyzer for Simulink models that can identify data and control dependencies between inputs, internal variables and the outputs of the system and the assertions are used to infer specifications of the system.

2.6.3.3 Invariants with continuous values

Quantitative association rule mining [83, 84] has been proposed to mine association rules from continuous variables. However the value ranges inferred by them were often not very accurate. The Range Miner on the other hand uses simple data binning and but then leverages our already existing iterative verification framework to refine the mined invariants.

The GoldMine tool [82] uses decision trees that can be used to derive quantitative association rules from the resulting trees. The applications are is the verification of hardware designs. However, by using single decision trees we often cannot infer rules that are true but have very low support and by using decision-forest-based methods, or gradient boosted trees (using multiple decision trees) the rate of false positive rules is very high. Furthermore, decision trees do not offer the very useful property of association rules that invalid rules will not be inferred again if one piece of evidence contradicts them.

Several approaches have improved the type of invariants that can be mined [61,85–87]. Nguyen et al. [61] proposes an approach that leverages equation solving, polyhedra construction, and SMT solving to infer polynomial and array invariants. They try to infer invariants that can help in program verification and are most interested in inference of loop invariants. They use a variety of rule templates that cover often used loop invariants (e.g. $x <= n1$). Garg [58] and Sharma [28] also present an alternative approach for the inference of array invariants. Li [85] extends the idea of invariant inference by introducing so called second order constraints, which are user-provided constraints that contain knowledge about invariants and can be leveraged to improve the quality of the mined invariants. The focus of requirements extraction in our work is to improve the approach so that the resulting invariants better relate to software requirements. For future work we are interested in evaluating other types of invariants such as the ones presented above and evaluate if they yield useful information about system behaviors. Sharma present the c2i tool [86] which uses randomized search to infer invariants. It describes the idea of using abstract properties to infer invariants for complex data like strings. The Specstractor tool also allows the user to define abstractions that are then applied to the data before the data mining step.

Chapter 3: Evaluations of Model-Based Testing

As part of this thesis, MBT was evaluated and analyzed in several case studies to determine its effectiveness and efficiency to identify ONBs, as well as its strengths and weaknesses and the effort that is necessary to introduce and apply it. This chapter gives an overview about three published case studies [26–28]. The overview presents the case study and the corresponding results. Each published version of the case study contains details of the MBT process that was used in the study. These details have been omitted in this chapter and are aggregated in the background section of this thesis as a general introduction to the MBT process (Chapter 2.1). The remainder of this chapter contains lessons learned with respect to applying the MBT process to test for ONBs and a discussion of what classes of ONBs can be identified using MBT.

3.1 The OSAL Case Study

The OSAL case study [28] evaluated the applicability and effectiveness of MBT for the identification of ONBs in mission-critical systems, which already underwent extensive testing. NASAs Operating System Abstraction Layer (OSAL) was the subject system. The OSAL is a reusable framework that wraps several operating systems (OS) and is used extensively in NASAs flight software missions. The system was modeled as a hierarchical finite state machine (FSMs) using the API documentation as the basis for the model creation. Any deviation from the API documentation and any extra behaviors not mentioned in the documentation are considered ONBs.

3.1.1 The System Under Test

The SUT in this case study is NASAs OSAL, which is an operating system abstraction layer used in space missions such as the Solar Dynamics Observatory and the Lunar Renaissance Orbit. In these applications, reliability is critical and hence OSAL has high quality requirements. OSAL ensures that programs behave in the same way on different OSes, allowing developers to switch the OS without changing the code. OSAL wraps the OSes POSIC, VXWorks6 and RTEMS and can be easily extended to include another OS. The development engineer selects which OS to use at build time. An overview of the OSAL architecture can be seen in Figure 3.1.

We applied MBT to the POSIX version of OSALs file system API. This API consists of two parts: osfilesys and osfileapi. The file osfilesys.c is responsible for handling the file system (i.e. creating, mounting, and un-mounting a file system etc.). The file osfileapi.c is responsible for the file and folder functionality (i.e. creating, copying, deleting files etc.).

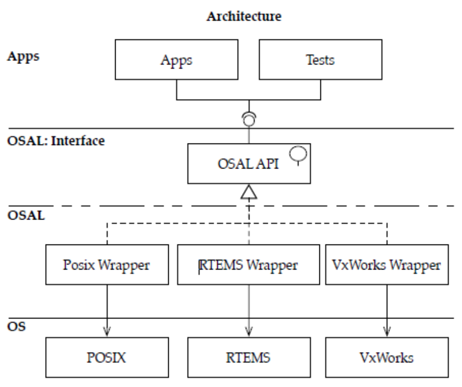


Figure 3.1: Overview of NASA's OSAL architecture

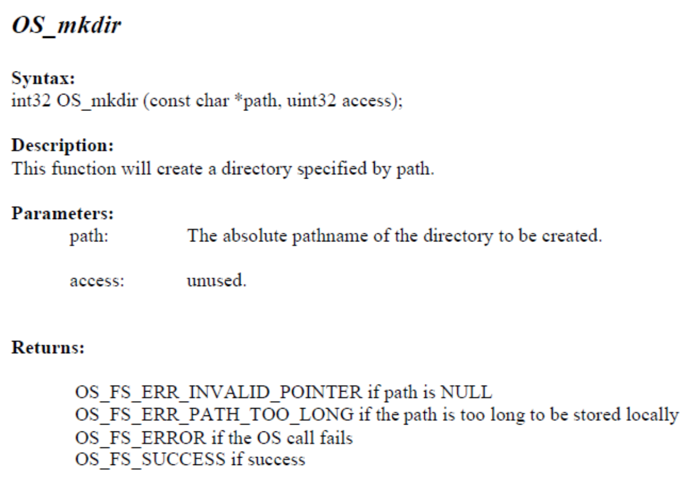
3.1.2 MBT applied to OSAL

The test execution framework was based on the programming language C in conjunction with NASAs UT_Assert testing framework. The input to the modeling effort was the API documentation that is distributed with the OSAL source code. It contains the syntax of each function, a description of its functionality and how it affects the overall system state, as well as its inputs and expected outputs. It also contains documentation about how it behaves for off-nominal inputs and shows corresponding error codes.

An example of the API documentation for the create folder function can be seen in Figure 3.2. We also analyzed the existing test cases of the system which helped us to understand how the system shall be used and to build the test infrastructure for the model.

3.1.3 Research Questions

In the case study we investigated the following four research questions:



OS_mkdir

Syntax:
int32 OS_mkdir (const char *path, uint32 access);

Description:
This function will create a directory specified by path.

Parameters:

path:	The absolute pathname of the directory to be created.
access:	unused.

Returns:

OS_FS_ERR_INVALID_POINTER if path is NULL
OS_FS_ERR_PATH_TOO_LONG if the path is too long to be stored locally
OS_FS_ERROR if the OS call fails
OS_FS_SUCCESS if success

Figure 3.2: API documentation for the create folder function

1. R1: Can MBT detect ONB in a fielded system that was already tested and reviewed extensively?
2. R2: Based on tests generated using MBT, how much code coverage can we achieve?
3. R3: Why could some parts of the code not be covered?
4. R4: What are the strengths and weaknesses of applying MBT to test APIs?

R1 asks the question if MBT can find additional bugs that were not found in the existing manual testing procedure. R2 and R3 are concerned with the code coverage that can be achieved with MBT and why parts of the system could not be covered. R4 then concludes the study with an analysis of the strength and weaknesses of applying MBT to a system like OSAL.

3.1.4 Case Study

We used the MBT approach based on finite state machines described in the background section 2.1 to build a model of 28 of OSALs functions based on the API documentation. Four functions were not modeled because one was not implemented, and even though the other three are part of the API, they were not directly related to the file I/O API functions that the modeling effort focused on. The resulting model has 274 states and 358 transitions as well as 100 high-level instructions and 100 assert statements. Thus, many of the transitions are used to transition between different parts of the model but do not have any associated action in the SUT. We used the model to generate two sets of test cases for the OSAL system. The first set of test case were generated using a transition coverage algorithm [22], that for each test case finds the shortest uncovered path from the start state to the exit state that together, across all test cases, achieve full transition coverage. In this case, we generated three test cases that together covered all transitions. This minimal set of test cases is a good start for testing since they cover the entire model. In addition, they are helpful for debugging the test execution framework since they ensure that each function in the API is called at least once. For the second set, we generated 1,000 test cases using a random walk algorithm. This algorithm traverses the model by randomly selecting transitions until the exit state has been reached. The test cases have an average of 299 high-level instructions. Since instructions are always paired with assert statements, the test cases have an average of 299 assert statements.

3.1.4.1 ONB Analysis

With the generated tests from our models, we were able to detect 11 new ONBs that were previously unknown. The following are descriptions of those ONBs. The ONBs are grouped into the three ONB types established in chapter 1.1. In some cases the same or a similar ONB was found in different places in the code. We used the practice of counting each instance of an ONB.

- **Bug/Defect:** The *copy file* and *move file* functions return success even if the source/target file (i.e. the file to be copied, or its intended target) do not exist. Furthermore the underlying operating system throws an error that is not caught by the OSAL but is only shown on the terminal during the testing process.

  The *copy file*, *rename file*, *make directory* and *open directory* return a generic error code for the file system when the names of the files are too long. However, the API specifies a specific return code for that case.

  The *rename file* documentation states that if the new name is too long it should return a specific error code, but the implementation actually never checks if the new name is too long.

  The *create file* and *create folder* functions sometimes ran into an issue where OSAL returned an error code that indicated that all file descriptors were full even though there should be free ones available. It turned out that the file descriptors are not removed properly. The OSAL keeps track of all created files and folders in a file descriptor table. The size of this table can be set by the OSAL user. After the file system is removed, the file descriptor table should be cleaned out. However, this never happens. The test case detected this issue by first creating several files which was successful, the test case then removed the file system which should have deleted the previously created file descriptors. The test case then added more files. Eventually the create file function returned an error code indicating that no more files could be created even though it should be possible.

- **Good Surprise:** There is a mismatch between the documentation and the implementation in the *create file*, *remove file*, *open* and *close file* functions. The implementation returns a specific error code when the path of the file cannot be parsed. This error code is not mentioned in the API documentation.

- **Bad Surprise:** We found no ONB in this category.

3.1.4.2 Coverage Analysis

The second part of the study was to understand how code coverage might relate to model coverage. As explained above, the model was created based on API documentation and other external descriptions of the SUT. Thus, the internal structure of the SUT was not used as a driver for modeling. However, code coverage is often an important testing criteria (in some environments, a certain level of code coverage is required) and it is therefore useful to understand the level code coverage that can be achieved through this form of black box testing. It is also useful to understand what prevents us from achieving 100% code coverage so that new code can be designed in such a way that code coverage can be maximized.

For the coverage analysis, we created six sets of test cases. The first set contains the test cases that were generated to achieve 100% transition coverage (and thus also 100% state coverage) of the model with a minimal set of test cases. This produced the three test cases that together achieve full transition coverage. The four other groups are random tests with 10, 25, 50, 100 and 1,000 test cases. These sets were used to compare how the coverage of the code increases as we increase the number of random test cases.

Figure 3.3, shows how the statement coverage changed when we added more test cases. For example, for the file osfileapi.c, the statement coverage for the minimal set of test cases is 87.12%. The 10 random tests also have one 100 percent transition coverage of the model. What sets them apart from the minimum coverage set is that they reuse transitions and therefore explore more paths through the model. This leads to 95.39% statement coverage, an increase of about 8%. Increasing the number of tests to 25 only yields a minimal increase of 0.34% to 95.73%. After that the statement coverage stagnates. The picture is different for the file osfilesys.c as the statement coverage here is 95.62% for all 6 sets of test cases because we only modeled those basic functions that are needed to run the file I/O API functions. Out of the 28 functions that were modeled 19 achieved 100% code coverage.

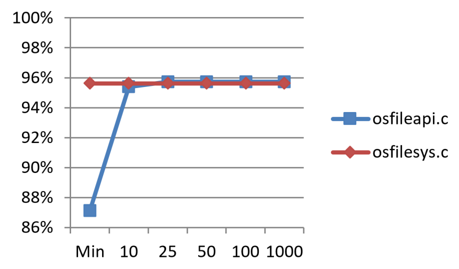


Figure 3.3: Statement coverage of the osfileapi.c and osfilesys.c. This represents the coverage after the model defects were removed.

3.1.4.3 Analysis of Code Coverage

We ran the generated test cases using GNUs gcov code coverage tool. The gcov tool reported code coverage statistics for each function. In this section, we characterize the reasons for not covering certain code statements during testing. Defensive programming, inability to trigger environment errors, and modeling errors are the three high-level reasons for not being able to cover all statements automatically.

1. *Defensive programming* Since OSAL is used in flight software missions, the functions of OSAL perform extensive error checking. For example, we found cases where the caller of a function checks the that the parameter values are valid before they are passed through a function call. In addition, the callee also checks the same input parameters in exactly the same way. In this scenario, it is only possible to cover error handling code at the callers side but not on the callees side because the callees error handling code will never be executed, it is dead code.

2. *Inability to trigger environmental errors* Since OSAL is a low-level software

package, it naturally deals with file I/O errors, memory errors, hardware signals and interrupts. However, these features are in conflict with code coverage because triggering errors such as hardware I/O error, file read or write error, out of memory error requires manipulating the environment. In a typical testing environment, it is difficult to trigger file read error unless we corrupt the physical memory blocks, for example. Thus, the code statements that deal with error handling of the underlying environment were not executed during automated testing. In order to trigger such errors requires an environment model that systematically introduces different types of errors, but that topic is beyond the scope of this paper.

3. *Modeling and data provider errors* Since modeling is a manual effort, some scenarios were either not correctly modeled or accidentally left out. Another source of error was not covering the data ranges for parameters. For example, we were opening files with read or write permission, but never in the append mode. We fixed those types of data provider errors based on the analysis of code coverage statistics. In Figure 10, the data refers to coverage after we fixed these modeling and data provider errors.

3.1.5 Strengths and Weaknesses of MBT on OSAL

3.1.6 Strengths of MBT

We identified the following strengths based on our experience with MBT on OSAL.

1. The scope of testing is crystal clear from the model. In contrast to cryptic scripts, which speak a technical language, the models are domain-oriented and easier to understand and reason about. For example, one may look at our OSAL models and understand what scenarios are tested and which ones are not tested without going through the test scripts.

2. Large number of test cases in a press of button. Once the model and the associated infrastructure are established, we can generate large numbers of unpredictable test cases, for example by random walks on the model. This addresses the problem with testing using manually created scripts, which tend to test the same code again and again.

3. Coverage of Corner-cases. Covering corner cases is an attractive capability of MBT due to its rigorous nature of traversing the model and ability to generate tests based on different model-coverage criteria. This ability should be compared to manually constructing test cases based on expertise and gut-feeling. Such manual test creation can typically not generate many enough test cases to detect corner case issues, which is not a problem for MBT.

4. Switching of test execution frameworks is very easy. We first generated test cases and ran them using the CuTest test execution framework. Later, we switched to NASAs own test execution framework called UT_Assert. We were able to make this switch and execute the generated tests without modifying the model. We only had to modify the template for header and declarations that were automatically inserted into every test case. In comparison, if the

tests are manually constructed, it would be very time-consuming to switch from one test execution framework to another because each line of code most likely would have to change.

3.1.7 Limitations of MBT

We identified the following limitations based on our experience with MBT on OSAL.

1. Generated test cases are difficult to comprehend. File, function and variable names of these randomly generated tests do not explicitly explain the actual scenario being tested. In addition, there is no high-level summary that explains each test. This makes generated test cases difficult to understand. In manual testing, test cases are typically commented up-front. For example, comments such as: This test verifies the scenario of deleting an existing file are difficult to obtain in the MBT context.

2. Finding the root cause of test failures is difficult. Long generated tests that fail can be difficult to debug. In many cases, we had to manually remove code in the generated tests to produce a small subset of scenarios that lead to test failures. This process can be time-consuming.

3. The same test failures in multiple tests are difficult to recognize. Tests are generated from a model and usually there are overlaps of transitions in one or more tests. Thus, if there are for example five test cases that fail, it does not

mean all these failures are independent issues. We had to manually analyze each test failure in order to confirm that they are indeed the same issues.

4. Programmers are not used to modeling. Programmers are used to writing test cases and source code but they hardly perform (abstract) modeling. Thus, MBT is a shift in the testing paradigm because it requires more abstract thinking. However, we have been observing that even under-graduate students are able to learn modeling using finite state machines. Thus, this limitation could be somewhat addressed through training and better modeling support.

3.1.8 Threats to Validity

The discussion about threats to validity is based on the model of Wohlin et al [88]. They define four classes of threats to validity, namely threats to internal, external, construction, and conclusion validity.

3.1.8.1 Threats to internal validity

Threats to internal validity are caused by factors that were not considered but might have influenced the results of the case study.

The results of this study indicate that the presented MBT approach is efficient in detecting defects in the SUT. However, if the defects were previously known by the test engineer who conducted the MBT, then the test engineer would have been biased and could have constructed the model in such a way that such known defects would have been exposed. In such a case, the results would be attributed to the knowledge of the test engineer rather than to the use of the MBT approach.

We view this risk as non-existent for several reasons:

1. The detected defects were not known to the OSAL team at NASA, thus there is no risk that the test engineer could have tailored the model to a certain set of known defects. In addition, the OSAL has been tested before as part of the regular NASA process and those detected defects were removed.

2. We have not conducted any other type of testing of the OSAL. Thus there is no risk that defects detected using other testing techniques were re-detected by the MBT approach.

3. We have previously analyzed OSAL using static analysis techniques and detected defects. However, a majority of those defects were of different types and would have been impossible to detect using MBT. There is some overlap of defects that would have been difficult, but not impossible to detect using MBT. For example, if the same test cases were executed against several OSes through their OSAL wrappers, it would have been possible to detect some of the defects we detected statically. For example, in a previous study [89], we reported defects related to differences in how OSAL OS wrappers use return codes. For example, one wrapper returned a generic return code (i.e. FS_OS_ERROR), while another wrapper returned a specific return code (i.e. FS_ERR_INVALID_POINTER) in a comparable situation. However, this particular defect was not detected using MBT so the risk is small that previous results influenced the modeling in a significant way.

4. An analysis of the model reveals that its structure is based on the API and on the structure of the code. Each API function is modeled in the same way, using the return codes as the driver. API functions that depend on each other (e.g. a file must be created before it can be deleted, unless the goal of the test is to test the response to evil requests) are modeled accordingly. Thus, the model is uniform and does not cover one area of the system more extensively than another. Thus, we deem the risk to be low that it was the test engineers knowledge about weak areas in the code that led to the detection of defects. Instead we believe it was the MBT approach that resulted in the detection of the reported defects.

3.1.8.2 Threats to external validity

Threats to external validity are concerned with whether we can generalize the results outside the scope of our study. There are several questions related to this, for example: 1. Would someone else, who performed MBT on the same system using the same approach arrive at the same results? 2. Would someone else, who performed MBT on another system (in a different domain, different language, different use of external entities, different types of software or technology platform) using the same approach arrive at similar results?

1. *Different test engineer* Experiences with using FSMs for MBT and basic programming knowledge are important in order for other test engineers to repeat and produce the same results presented in the case study. We have instructed

others on how to use this MBT approach and since they arrived at similar results, we have strong reasons to believe that another test engineer also would produce similar results. In fact, the principle of modeling each possible return code as a transition, results in a predictable number of transitions. In addition, the way the states are used to capture the conceptual states of the SUT results in a predicable structure of the model. However, since the models can be divided into sub-models in a somewhat subjective manner, still with the same overall functionality, we do not expect the number of sub-models to be exactly the same. Since test cases are derived from the model by Jumbl, we do believe that another test engineer using the presented modeling approach would detect the same issues in OSAL. If the same approach was applied to another SUT, we believe that the same types of issues would be detected, i.e. differences between expected and actual behavior based on return codes and crashes.

2. *Different domains* The test engineer who conducted MBT on OSAL did not possess domain knowledge of flight software and how it might use the OSAL in detail. The test engineer has not been involved in the development or maintenance of OSAL. Thus, we do not see any reason why the proposed method would not be repeatable on software systems in different domains as long as they have an API. However, it is critical that the functions of the API return values that reflect the outcome (success and failure codes) of each function call. If this is not the case, then the API must offer other functions

that can be used to provide the same information. Without such visibility into the state of the system, any form of black box testing through the API will be difficult.

3. *Different programming languages* The modeling approach is independent of the programming language of the SUT as well as of the test execution framework. The actions associated with the transitions are high-level instructions that are replaced by programming statements outside of the model. Thus, these abstract test cases derived from the model can be translated into any programming language and test execution framework without changing the models. Thus, as long as the programming language supports the notion of functions and return codes, as described above, the risk that the presented approach would not work on systems written in other programming languages is small.

4. *Different types of software, technology platforms* The approach of using MBT to test a system through its API can in fact be used on most types of systems independent of the software architecture or technology platform. For example, the GUI of a web-based system can be viewed as an API. Thus, the behavior of such as system can be modeled using the same approach, and the same type of abstract test cases can be derived from the model. The only difference would be in the test execution framework (e.g. Selenium[1]), which would be used to navigate the GUI and check that the actual state of the GUI matches

[1]http://www.seleniumhq.org/

the expected state (e.g. the correct web page has loaded as an effect of clicking the submit button). This is based on that fact that we have conducted successful (but unpublished) studies where the approach was applied on web-based systems. There is a risk, however, that this approach does not work on some embedded systems that lack a regular API. If they are dependent on a certain hardware platform, it would be even more difficult to test such a system. At the same time, any type of testing of such systems is difficult. When it comes to the code coverage, the results are probably not transferable to another system, unless it is very similar to OSAL. Since the OSAL mainly consists of wrapper code, it only translates from one interface to another and therefore is relatively limited in complexity. This is most likely why the code coverage reached about 96%, which is very high compared to typical results.

3.1.8.3 Threats to construction validity

Threats to construction validity assess if the correct measurements were used in the case study. In this study we used the number of issues as the basic measurement. This is a direct measure that is not derived and therefore should be a good indicator for the effectiveness of the MBT approach. In order to validate that the detected defects were indeed relevant, the NASA OSAL team reviewed each issue and determined that this was the case.

3.1.8.4 Threats to conclusion validity

Threats to conclusion validity cover issues that affect the ability to draw the right conclusions from a case study.

Our conclusion is that the presented MBT approach is effective in detecting defects in OSAL and systems similar to OSAL. Considering that it is easier to detect defects in a low-quality system (because there are more defects to be detected), it is important to remember that the OSAL has been thoroughly tested as part of the regular NASA development process and already had high quality when this study started.

In addition, we conclude that there is a relationship between model coverage and code coverage that was measured to be relatively high. However, in order to maximize coverage, the testing model need to be able to manipulate the environment the system is running in (i.e. restrict the available memory on disk, in RAM, restrict network access, introduce time delays etc.).

The study was done on only one system, which restricts the generalization of the study. Nevertheless it can be compared to other studies and be a valuable data point for comparison with future studies. In addition, as explained above, we have conducted similar studies on other types of systems with similar results regarding defect detection. Thus we believe the risk is low that the conclusion is incorrect.

3.1.9 Conclusion

We developed a suite of behavioral models represented as hierarchical FSMs, of OSALs core file system API and generated a large number of test cases automatically. We then automatically executed these test cases against the OSAL. The results show that the OSAL is a high quality product. Due to the systematic and rigorous nature of MBT, we detected a few previously unknown ONBs, which escaped traditional manual testing and code reviews. Thus, we conclude that MBT is both applicable and effective for testing system such as the OSAL and is especially effective in detecting those types of ONBs that crash the system or can be observed by comparing expected return codes with actual ones.

We discussed the MBT architecture, the analysis of detected ONBs, and code coverage of generated tests. The generated test cases achieved high code coverage (about 96%). We also found that defensive programming and low-level error checking features (e.g., file I/O errors) make it difficult to achieve 100% code coverage unless the environment can be manipulated so that the application runs out memory, disk space etc. We also showed that maximum code coverage can be achieved with a limited number of test cases and that adding more test cases did not increase code coverage. However, since some of the detected defects required very long and repetitive test cases, we conclude that code coverage is a necessary but unsatisfactory testing metric.

We also discussed the strengths and weaknesses of MBT. We conclude that that it would be beneficial to overcome the identified weaknesses, such as facilitating the analysis of failed test cases and to help the tester with the model creation process, for MBT to be an undisputed technology candidate for infusion into everyday practice.

3.2 The GMSEC Case Study

The GMSEC Case Study [90] was originally published as a short abstract [27] and conducted together with Vignir Gudmundsson who was a visiting scientist at Fraunhofer CESE during the study. The full study [90] has also been published as part of his Master's Thesis at the University of Reykjavik [91].

The SUT of this case study was the software bus of NASAs Goddard Mission Service Evolution Center (GMSEC). The goal was to study the feasibility of using extended finite state machine (EFSM) based MBT to identify ONBs in a real-world software system that was designed to be flexible. GMSEC has three levels of flexibility: 1) loose application coupling through a software bus based on the publish-subscribe architectural style, 2) language independence by providing APIs to the bus in several programming languages, 3) middleware independence by providing wrappers for several middlewares that are supported by the software bus.

The novelty brought forward in this study is that one model and one set of generated test cases was used as the basis to test the software bus for behavioral consistency across multiple programming languages and middleware wrappers.

3.2.1 The System Under Test

GMSEC is a flexible and reusable framework that provides a scalable, extensible ground system for current and future missions both inside and outside NASA. The built-in flexibility has allowed many space missions (e.g., the Global Precipitation Measurement (GPM) mission, the Magnetospheric Multiscale (MMS) mission, and the Lunar Reconnaissance Orbiter (LRO)) to build their systems based on GMSEC. A GMSEC application is a large component that accomplishes a certain set of functionality. For example, the GMSEC System Display (GEDAT) application provides a visual representation of the GMSEC environment and can display data system activity at any level within the network.

The bus allows GMSEC-compatible applications to be integrated with the system in a plug-and-play manner. There are many GMSEC-compatible applications (application for short) on the market already, thus one can quickly build the basis for a new ground system in a few hours. In case the necessary application is not already available, it can be implemented in one of the supported programming languages. By adding the GMSEC API for the applicable programming language, the application becomes GMSEC-compatible and can be plugged into any system that is based on the GMSEC bus. In addition to the API that is supported in different programming languages, there are several different middleware products available on the market that are supported by GMSEC. The applications that use the GMSEC API do not access the middleware directly but through a middleware wrapper. The software bus implementation redirects all method calls to appopriate middleware wrappers at runtime. There is one wrapper for each middleware product. While middleware products typically handle the transportion of data, the software bus adds messaging based on the pub-sub style [92] in a consistent way across all supported middlewares.

Testing of the bus is a complex endeavor because of the flexibility it offers. The bus offers a set of commands related to the pub-sub style that must be tested. However, the user has great flexibility in creating valid command sequences making it a challenge to test them all.

In addition, the system supports a flexible number of subjects to which applications can publish, where subscriptions can be formulated using regular expressions. Since the API is implemented in several different programming languages, each such API has to be tested. The fact that there are wrappers for many middlewares, which all must be tested, adds to the testing burden. Ideally, there is a suite of test cases that cover all possible combinations of valid pub-sub commands. Ideally, the test suite would then be automatically translated to each of the supported programming languages, and executed on each of the supported middlewares, thus indirectly testing each wrapper.

Over the years, the GMSEC team has developed a set of executable test cases that achieve some of these goals. However the test suite is hand-written and limited to a couple of common pub-sub scenarios. In addition, the test suites for the different programming languages and middleware wrappers are not identical and thus the results cannot easily be compared across configurations. Naturally, the GMSEC team conducts large amounts of testing. They also continuously review the artifacts in order to detect potential issues. In addition to their rigorous reviews and testing, the GMSEC team prefers an independent organization such as Fraunhofer CESE to review the implementation quality [93] as well as to evaluate new technologies for potential infusion into the GMSEC project.

3.2.2 Nominal Behaviors of a Message Bus

In the publisher-subscribe architecture there are four key concepts: publishers, subscribers, messages, and subjects. The nominal behaviors of the associated actions of these concepts are as follows. Publishers and subscribers communicate indirectly via the message bus (bus for short). A subscriber can subscribe to subjects and any message published by a publisher to the bus that matches the subject is routed to the subscriber. Publishers publish to the bus and it is ultimately the bus' responsibility to route the message to all subscribers. The messages are placed into a message buffer (akin to a mailbox) that the application can periodically check. In GMSEC this buffer holds messages from all subjects. However, in other publish-subscribe style applications that we tested at Fraunhofer [94] there could be several buffers depending on the number of subjects an application is subscribed to.

Both publisher and subscriber are unaware of each other and are thus loosely coupled and only aware of their subjects and messages they send or receive. An application can be subscriber, publisher, or both. If an application is subscribed to a subject and sending out a message to said subject it will also receive a copy of the message. Before an application can do anything it first has to connect to the software bus by creating a connection object and then calling the connect command. This will make the software bus aware of the application and it can now start publishing and/or subscribing.

There are also a number of other NBs related to the sequencing of actions that are important to describe as rules because they dictate the type of testing that needs to be performed.

1. **Establishing a connection to the bus:** An application must be connected to the bus in order to interact with it. An attempt to connect if the application is not already connected should succeed, otherwise it should fail.

2. **Subscribing to a subject:** An attempt to subscribe to a subject that the application does not already subscribe to should succeed, otherwise it should fail. Unsubscribing to a subject: An attempt to unsubscribe to a subject that the application already subscribes to should succeed, otherwise it should fail. A successful unsubscription to a subject removes the subscription for that subject, thus the application is allowed to subscribe again to the same subject.

3. **Delivery of messages:** In order for a message to be delivered, an application must subscribe to the subject before messages of the same subject are published. Only if this is the case, the published message is delivered to the subscribers buffer. Once the message is delivered to the buffer, the subscriber can retrieve that message at any time.

4. **Reliability:** The bus should deliver one copy of each message to each subscriber, once and only once.

5. **Clean up upon disconnect:** When an application disconnects from the bus, the bus should remove all of the applications subscriptions as well as all of the applications messages in the buffer that have not yet been delivered. Thus, if the application connects again to the bus, it has no subscriptions and no messages.

3.2.3 Testing Questions

From the above mentioned NB, we derive the following testing questions, which together could be used to drive the modeling and the testing of any bus that is based on the pub-sub style.

1. Q1: Are there ONBs related to the connect/disconnect functionality?
2. Q2: Are there ONBs related to the disconnect cleanup functionality?
3. Q3: Are there ONBs related to an application subscribing to one or more different subjects?
4. Q4: Can a component subscribe to the same subject multiple times without unsubscribing in between?
5. Q5: Can a component unsubscribe to the same subject multiple times without subscribing in between?
6. Q6: Can a component unsubscribe to a subject it is currently not subscribed to?

7. Q7: Are there any missing messages?

8. Q8: Are there any extra messages?

9. Q9: Are there ONBs related to the validation of message subjects?

3.2.4 Case study

Since the GMSEC team is interested in improving their testing processes through automation, the goal of the study was to study the feasibility of using MBT to identify ONBs in GMSEC. Another goal was to evaluate extended finite state machines (EFSMs) as the testing models to identify their strengths and weaknesses. State machines were chosen because they are easily understood and open source MBT tools that generate test cases from them are readily available [95].

A major question for the study was: Can the GMSEC software bus be modeled in such a way that test cases can be automatically generated to test the software bus and its publish-subscribe messaging features as well as all possible combinations of programming languages and middleware wrappers without modifying the model? Since GMSEC was already tested by the GMSEC team, the MBT approach would be viewed as successful if it were able to detect ONBs that had not been uncovered by those earlier testing efforts. This study covers the testing of the GMSEC software bus and its abilities to perform messaging according to the publish-subscribe style, thus no GMSEC applications were modeled or tested in this study. Instead, the test cases are generated from dummy applications that use the software buss APIs in different but realistic ways.

3.2.4.1 The scope of the study

The GMSEC supports several programming languages (C, C++, Java, Perl). For this study we decided to test the C, C++, and Java APIs because they are the most frequently used ones. GMSEC also provides different middleware wrappers for commercial and open-source middlewares such as WebLogic, WebSphere, Tibco Smart Sockets, and ActiveMQ. In addition, GMSECs own middleware implementations Bolt and MB (a.k.a. Magic Bus) are provided as part of the GMSEC deliverable. These middlewares are supplying the underlying framework that realizes the message bus.

We decided to test the wrappers for WebLogic, Bolt, and MB in this study. Because of commercial licensing issues, other middleware wrappers were excluded. It should be noted that while the commercial middlewares have already been tested by their vendors, the wrappers were developed as part of the GMSEC project and therefore must be tested by the GMSEC team. This type of testing is provided as part of our application of MBT to GMSEC. The study was carried out independently of the GMSEC team, but questions were asked to the GMSEC team during the process when the documentation was not specific enough. The findings were reported to the GMSEC team at the end of the study.

3.2.4.2 Applying MBT on the GMSEC API

Modeling of the GMSEC bus was driven by the specifications for the core features of a software bus, the associated rules, and the derived testing questions

Table 3.1: Results of the GMSEC ONB Analysis

ID	Testing Question	Answer	MWW	API
Q1	Are there ONBs related to the connect/disconnect functionality?	Yes	Yes	No
Q2	Are there ONBs related to the disconnect cleanup functionality?	Yes	Yes	No
Q3	Are there ONBs related to an application subscribing to one or more different subjects?	Yes	No	Yes
Q4	Can a component subscribe to the same subject multiple times without unsubscribing in between?	No	No	No
Q5	Can a component unsubscribe to the same subject multiple times without subscribing in between?	Yes	No	Yes
Q6	Can a component unsubscribe to a subject it was never subscribed to?	No	No	No
Q7	Are there any missing messages?	Yes	Yes	No
Q8	Are there any extra messages?	Yes	Yes	No
Q9	Are there ONBs related to the validation of message subjects?	No	No	No

that we introduced earlier. The model was then used to generate abstract test cases, which were automatically translated into three sets of concrete test cases, one for each programming language that we chose. Each set was then executed on each of the three middlewares. Thus, the abstract test suite was used to test nine different concrete configurations. We created 100 test cases that each achieved full transition coverage of the model.

3.2.4.3 Results

The summarized results of the ONBs from executing the 100 tests in the nine versions are shown in table 3.1. The answer to the testing questions is provided in Column 3. Column 4 indicates whether the issues could be traced back to the middleware wrappers (MWW), whereas Column 5 shows if there where inconsistencies between the different GMSEC API implementations (Java, C, or C++).

The tester detected ONBs for six of the nine testing questions. Four of the

ONBs could be traced back to the middleware wrapper and two to inconsistencies in the APIs. We only detected ONBs in the Bug/Defect category, none were found in the Good or Bad Surprise category.

One of the ONBs we encountered in the connect/disconnect functionality (Q1) manifested itself only on the Bolt middleware, when the generated test cases called the connect/disconnect functionality several times in short succession. After several consecutive calls the connect function would start to return an error code. Another ONB that is middlware specific was in relation to Q8 (multiple messages) and occured for the middlewares Bolt and WebLogic. The ONB was that an application could retrieve several copies of the same message when in fact there was only one. When the test case was executed on MB, the message could correctly only be retrieved once, but when it was executed on either of the other two middlewares, the application was able to incorrectly retrieve extra copies of that same message whenever it checked for new messages. This ONB is subtle and difficult to detect since, by design, applications are unaware of the number of messages that are sent to them at any given time. Thus, this ONB could cause problems if the retrieving application does not check for duplicate messages. An application that was developed and tested using MB would, for example, not have problems with duplicate messages and therefore most likely would not check for duplicate message. However, if MB is switched for Bolt or WebLogic, which is effortless due to the built-in flexibility, problems with duplicate messages would suddenly occur without the programmer necessarily understanding why, since the application passed all tests and worked without problems in the field.

An example of an API inconsistency is an ONB we encountered in relation to Q3 (subscribing several times). The Java implementation of the API does not handle the differences between a regular subscription and a callback subscription correctly. As mentioned before, a regular subscription and a subscription with a callback to the same subject should be treated as two different subscriptions. The problem occurred in situations where a regular subscription was followed by a subscription with a callback to the same subject. Both operations should return success. However, the Java implementation incorrectly returns an error for the subscription with callback.

We were also experimenting with modeling GMSEC as a FSM model, however the FSM model was not well suited for modeling multiple subscriptions and multiple messages and it was therefore not possible to answer all testing questions using a FSM model.

3.2.5 Lessons Learned

Here we present our lessons learned from the case study of applying MBT on GMSEC. Many of these lessons are independent of the pubsub style and are also relevant for testing of systems based on other styles.

- **Existing test cases are valuable assets for MBT.** The tester first reviewed the existing unit test cases in order to build a model and also to define the mapping between actions of the model and concrete APIs to interact with the SUT. Without the existing test cases, it would have taken a lot more effort to perform end-to-end MBT because independent testing teams may not know how to use the API. It is recommended to leverage existing test cases for introducing MBT.

- **Use an incremental end-to-end approach.** Executable test cases were generated, after modeling just two basic functions (connect and disconnect). Those test cases were run to make sure the most basic functions worked well. Afterwards, in each increment, additional behaviors were modeled and automatically derived and executed test cases and reported bugs to the GMSEC team. In some cases, the model and resolved abstractions that were used in previous increments we also refined. For example, the initial models abstracted the subject pattern by ignoring regular expressions. This abstraction is correct but incomplete, thus, later versions of the models introduced regular expressions. An incremental approach to MBT is very attractive, meaning that managers and engineers see the value of MBT immediately instead of waiting for the model to be completed.

- **Embed debugging information in the generated test cases.** When a generated test case fails, the tester has to review the test and understand the conceptual state of the system. For example, if unsubscribe fails, the tester has to manually reconstruct the conceptual subscription table. This process can be time consuming especially if the generated tests are lengthy. Debugging information such as the conceptual state of the subscription table, which is created during the model traversal time, was embedded into the generated test cases. The debugging information is of great value to understand the root cause of test failures and characterize the test failures to developers. Adding the debugging information will make the tests look more complex, but the benefits are overwhelmingly useful to reason about test failures.

- **Maintain traceability links.** When a generated test case fails, the tester knows the line number where the assertion failed. However, the sequence of actions that led to the failure is often not immediately clear. It is time consuming to walk through the concrete test case in comparison to the abstract test case because the former is very detailed (e.g., it includes a lot of set-up code, declarations, etc.) but the latter is often much shorter without any implementation details. However, it is not easy to map the line number of the test failure to the abstract test case. For this purpose, traceability links were maintained using the mapping data between high-level actions of models to low-level actions of the API, especially when tests fail traceability links are helpful. In the GMSEC case, the mapping data was used to maintain traceability links among the generated test cases for different programming languages that we support.

- **Models benefit from naming conventions.** One benefit of MBT is that the model is a precise documentation of the SUT. However, it is important to ensure that state names and transitions clearly explain the intention and follow a consistent naming convention. Such models help others to understand the scope of what is tested and not tested, identify weaknesses in the model, etc.

3.2.6 Conclusion

In this case study MBT was applied to the API of the software bus of NASAs GMSEC. The goal of the case study was to study the feasibility of using MBT to identify ONBs in a software system that is designed to be flexible. The premise of MBT is to provide a systematic way to test systems in an automated manner and the case study was conducted in order to evaluate whether this premise holds. The case study demonstrates that it is feasible to use MBT for a system like GMSEC. This was demonstrated, for example, by the fact that the test cases were able to detect ONBs in GMSEC, which is an already tested system. It also showed that it was feasible to use the same model to test different characteristics of the publish-subscribe architecture for different middlewares as well as different GMSEC APIs implemented in different programming languages without changing the models or abstract test cases.

We also found that MBT does provide a systematic way to test systems, and even though there are still manual steps involved, a high degree of automation is possible to achieve with reasonable effort. Although, in this paper, the testing approach was applied on a system based on the publish-subscribe style, it can be applied on any system suitable for MBT. Based on the lessons learned from this study, we have developed a tool for MBT that facilitate the use of Graphwalker for MBT as described in this paper.

3.3 The Elexnet Case Study

The Elexnet case study [26] compares manual testing as performed by a tester at Global Net Services (GNSI, one of our industrial partners) with MBT which was performed by a tester from the Fraunhofer Center for Experimental Software Engineering. The SUT is a web-based data collection and review system for publications about food-borne illnesses. The SUT was inherited by GNSI from another company. Under a federal contract by the Food and Drug Administration (FDA), GNSI then modified the SUT to meet new requirements.

3.3.1 Empirical Study

To be able to understand the impact of a new technology, one has to compare the new technology to the currently used ones, and collect relevant and comparable data. Naturally, in the context of GNSI, one would like to compare the performance of two (or more) experienced GNSI employees. However, since it would be too expensive and time consuming to first train a regular GNSI employee in MBT and then have that employee use MBT on a set of other systems to become experienced, we decided to rely on an outside tester for MBT with the drawback that such an outsider is not a domain expert and not familiar with GNSIs systems.

3.3.1.1 Study Goal

The goal of this study is to evaluate the costs and benefits of MBT based on FSMs as compared to completely manual testing, in an industrial setting at GNSI.

We designed the study so that we could compare the effectiveness (number and type of ONBs found) and efficiency (effort spent) of the two techniques.

3.3.1.2 Study Design

The study was carried out by two management teams. The GNSI management team coordinated all efforts on the GNSI side, appointed the manual tester, provided all artifacts (requirements, URL to test system etc.) and selected the SUT and the versions to be tested. The Fraunhofer management team coordinated all efforts on the Fraunhofer side and appointed the MBT tester.

The testing tasks were carried out by two testers. The GNSI tester, who used GNISs regular testing practices, was a GNSI employee with seven years of testing experience, of which the last 3.5 years were spent at GNSI. The manual tester has limited development experience and is not a programmer.

The Fraunhofer tester was a Fraunhofer employee with two years of programing and testing experience and one year experience of MBT during which this tester had tested three other web based system using MBT/Selenium as well as one API-based system. The MBT tester also had some support from a senior scientist at Fraunhofer to set up the testing architecture as well as from a student intern who helped develop the testing framework. The testers had no contact with each other during the testing process. The testers had access to identical documentation and could also contact the development team with questions about the SUT and its requirements, which they did on a couple of occasions to clarify how the NB of the SUT was supposed to work.

Each tester was instructed to apply their approach on two versions of the SUT with the goal to detect as many ONBs as possible. No time limit was given; the testers used their own discretion to determine when to stop each testing task while recording the time they spent. The second testing session (version 2) was observed by a Fraunhofer researcher who also followed up with interviews of the two subjects on their background etc. The observation was conducted in order to get a deeper understanding of the two different testing processes and where the effort was spent. Based on this data we collected lessons learned about the testing processes and identified improvement points in the workflows.

The SUT and the two different versions were selected by the GNSI study management group, who picked the SUT because it was representative of the types of systems GNSI develops. Thus the two testers did not have any influence on the system or version selection process. The testers did not have any previous knowledge of the SUT. The versions were tested as they were developed by the development team so to mimic a regular software development project. Thus, the testers only had information about the version they were currently testing. When they tested the first version, they, knew that a second version would be tested later, but they did not have any other information about the second version such as if any features had changed or what defects had been fixed. Since the study required that the two testing teams were relatively synchronized in their testing (i.e. that their testing was carried out during the same week), the study was conducted in parallel to the regular testing effort of the development and testing team.

None of the two testers interacted with the development team or the original testers beyond the initial questions about the requirements.

3.3.1.3 The System Under Test

The SUT is a web-based data collection and review system for publications about food-borne illnesses. The SUT was inherited by GNSI from another company. Under a federal contract, GNSI then modified the SUT to meet new requirements. The SUT has three different user roles: 1) *Scientists* can enter data and review and edit their own entries; 2) *Subject Chairs* can review and approve or reject the entries of subordinate Scientists; 3) *Managing Editors* have access to and can edit all entries.

The two testers tested two versions (V1 and V2) of the SUT. V1 is based on Oracles Portal framework and had been rolled out to the users for evaluation. V2 is a redesigned version where the graphical user interface (GUI) had been changed and was no longer based on the Oracle Portal framework. In addition, several issues that previously had been reported for V1 had been fixed. No new functionality had been added. Thus, V2 needed to be tested to ensure that previous defects had indeed been removed and no new defects had been introduced. In addition, since the GUI had changed drastically, it was important to test that no new defects had been introduced due to this change.

3.3.1.4 Artifacts available to the Testers

The following artifacts were available to the testers:

- **Use cases:** The testers were instructed to test the SUTs implementation of two use cases related to data entry and edit. Each use case description was about 1,600 words long.

- **Requirements:** Details of the two use cases to be tested were provided in the requirements specification. The detailed requirements document was 3,800 words long.

- **Access to developers:** The testers could contact the developers to discuss the use cases, the requirements or the SUT.

- **The SUT:** The testers had access to the running SUT through three user accounts, one for each role, but not to source code.

3.3.1.5 Data Collected

We collected self-reported effort data, i.e. each tester recorded the number of hours worked on a certain task. We also collected self-reported ONBs, i.e. each tester documented each detected ONB. The data was collected for each of the two versions.

3.3.2 The two Testing Processes

3.3.2.1 Manual Testing at GNSI

The following is a summary of the testing process at GNSI as it was conducted by the manual tester participating in this study.

In the first step, the manual tester builds an understanding of the SUT. For this, the tester analyzes the documentation (e.g. requirements, use cases), explores the SUT, and discusses the SUT with its developers. This step is necessary for the tester to understand how the SUT works.

In the next step, the manual tester writes down test instructions for the test cases. The test cases specify the actions that the tester should perform and the expected results, and are documented in an excel spreadsheet. Thus these test cases are a set of manual instructions on how to test the SUT. The manual tester uses them as a guide while testing the system. During this manual execution, the tester enters data, clicks on buttons, observes the resulting behavior, compares to expected behavior, and documents ONBs. The manual tester also does some exploratory testing beyond what is specified in the test cases. The steps involved in is type of exploratory testing are typically undocumented unless ONBs were detected.

3.3.2.2 Model-Based Testing at Fraunhofer

The model-based tester in this case study followed the approach described in Section 2.1. The system was modeled as a finite state machine FSM and tests were generated using the JUMBL tool [96]. The stopping criterion for Jumbl equals reaching the state labeled Exit. The tester can instruct Jumbl to either randomly traverse the model until the exit state is reached, or to produce as many test cases as needed to cover all states and transitions.

The test execution framework is based on Selenium[2], which automates testing of web systems. Seleniums Java API was used to interact with the SUT. JUnit[3] is the basis for the generated test cases and JUnits runtime environment executes the test programs.

The test generation and test execution process is automated and requires nearly no human effort. The test execution results in a list of test cases that passed or failed, and the location of the failure. A failing test case is analyzed by the tester to determine 1) whether it failed because of an issue in the SUT or of other reasons, and 2) how to reproduce the failure. This involves replaying the test case automatically or manually.

3.3.3 Results and Observations: Issues

3.3.3.1 ONB Classification

Each ONB was classified according to a scheme that was defined bottom-up i.e. it was based on the detected ONBs. Each class has an example associated with it based on an actual detected ONB.

[2]http://docs.seleniumhq.org/

[3]http://junit.org/

- **Business Logic:** ONB related to functionality. E.g. The user wants to save entered data but the save button is missing, thus data cannot be saved. The issues in this category are of the Bug/Defect ONB category. (Example: Business Logic #3)

- **Field Validation:** ONB caused by missing or incorrect field validators. E.g. Required field *method type* can incorrectly be empty. These ONBs also fall into Bug/Defect ONB category. (Example: Field Validation #1)

- **Extra Fields:** Inconsistencies between specified and actual fields. E.g. the specification does not list a field for phone number, but the GUI has one. These fall into the Good Surprise ONB category, since they are valid behaviors that are not properly documented. (Example: Extra Field #2)

- **Naming Discrepancies:** Inconsistencies between specified and actual labels. E.g. Specification lists an *Organization* label but label on GUI reads *Lab or Organization*. Since they are deviations from the documented behavior they are also in the Bug/Issue ONB category. (Example: Naming Discrepancy #1)

- **Usability Issues:** ONBs related to usability that do not interfere with the functionality. E.g. the layout is broken but the SUT is still fully functional. Since the layout has not been specified and the ONBs had detrimental effects on the usability these issues fall into the Bad Surprise ONB category. (Example: Usability Issue #2)

Table 3.2: ONBs detected in V1

Category	**Manual**	**MBT**	**Union**
Business Logic	9	19	22
Field Validation	1	6	6
Extra Fields	6	1	6
Naming Discrepancies	5	0	5
Usability	0	3	3
Total	21 (13+8)	29 (8+21)	42 (13+8+21)

3.3.3.2 Issue Analysis

The results from testing V1 are as follows: the Manual tester reported 21 ONBs and the MBT tester reported 29 ONBs, see Table 3.2. The combined number of detected ONBs (the union of the two sets) is 42. The overlap of the findings is small (8) and each tester reported several distinct ONBs (Manual 13; MBT 21), i.e. ONBs that were detected by one tester and not the other, see Table 3.3. The Manual tester reported more ONBs related to Naming and Field Discrepancies, whereas the MBT tester reported more ONBs related to Business Logic, Field Validation and Usability. It should be noted that all usability ONBs reported by the MBT tester were detected during exploration. No modeling related to Usability was done and therefore no usability ONBs were detected by MBTs test programs.

Table 3.3: Number of distinct ONBs in V1

ONBs detected only by Manual	**ONBs detected by both**	**ONBs detected only by MBT**
13	8	21

The results from testing V2 are as follows: the Manual tester reported 17 ONBs and the MBT tester reported 29 ONBs, see Table 3.4. The combined number of reported ONB (union) is 36, see Table 4. The MBT tester still reported more ONBs related to Business Logic and Field Validation, but for Usability, the two approaches reported almost the same number of ONBs (5 vs. 6) with a great overlap. It should again be noted that all usability issues reported by the MBT tester were detected during the exploration phase. Since the locators to automatically locate fields and buttons had changed significantly from V1 to V2, the MBT tester first had to explore the new version of the SUT in order to understand exactly how to address that change. In that process, new usability issues were detected. It is important to note that the ONBs related to Naming and Field Discrepancies that were reported for V1 were still present in V2, but were not reported by the manual tester this time.

The ONBs that were reported for V2 are still distinctly distributed among the two approaches: 19 of the 36 ONBs were reported only by the MBT tester and 7 only by the manual tester. The 10 remaining ONBs were reported by both testers, see Table 3.5.

We will now analyze the detected ONBs across the two versions. Since some

Table 3.4: ONBs detected in V2

Category	**Manual**	**MBT**	**Union**
Business Logic	10	17	22
Field Validation	1	5	5
Extra Fields	1	1	1
Naming Discrepancies	0	0	0
Usability	5	6	8
Total	17 (7+10)	29 (10+19)	36 (7+10+19)

Table 3.5: Number of distinct ONBs in V2

ONBs detected only by Manual	**ONBs detected by both**	**ONBs detected only by MBT**
7	10	19

Table 3.6: Number of total ONBs detected in the two versions

Category	**Manual**	**MBT**	**Union**
Business Logic	12	22	27
Field Validation	1	6	6
Naming Discrepancies	5	0	5
Extra Fields	6	1	6
Usability	5	7	9
Total	29 (17+12)	36 (12+24)	53 (17+12+24)

of the ONBs were fixed from V1 to V2 and since some new ONBs were reported, we summarized (using the union operator) all reported ONBs, see Table 3.6 and Table 3.7. The tables show the total number of ONBs found in the two versions, including all ONBs found in V1 plus all newly reported ONBs in V2. That is, an ONB that was reported twice by a tester, i.e. in both V1 and V2, is counted only once. The analysis show that the MBT tester found more ONBs related to the Business Logic and Field Validation categories, whereas the manual tester reported more ONBs related to the Naming and Field Discrepancy categories. The two approaches reported similar number of ONBs (Manual: 29; MBT: 36), see Table 6, even though they reported different ONBs (Only Manual: 17; only MBT: 24), see Table 3.7.

Table 3.7: Number of total distinct ONBs in the two versions

ONBs detected only by Manual	**ONBs detected by both**	**ONBs detected only by MBT**
17	12	24

3.3.3.3 ONBs Severity

In order to compare the value of the detected ONBs, we classified each issue using a severity score scale consisting of five levels that was defined by NASA [97], see Table 3.8. For simplicity, we used the inverse severity level as the severity score. Thus, issues on severity level 1 received a severity score of 5. ONBs on severity level 5 received a severity score of 1. ONBs related to business logic received scored between 1 and 4 depending on how severe each ONBs was; field validation ONBs all scored 2, and all other ONBs scored 1. By summarizing all severity scores, we see that Manual resulted in a severity score of 46 while MBTs score was 73. Thus, MBTs score was about 60% higher than manuals.

3.3.3.4 ONB Discussion

One observation is that only a few of the ONBs were found by both approaches. The mostly disjoint ONBs indicate that the testers focused their efforts on different aspects of the SUT even though they had the same testing goal (to detect ONBs related to the two use cases). The manual tester, for example, tested for discrepancies in the field names (a specific manual test case listed all expected field names).

Table 3.8: Overview of the results of the ONB Severity analysis

Severity Score	**Issue Type**	**Manuals Score**	**MBTs Score**
Severity 1: Prevent the accomplishment of an essential capability; or jeopardize safety, security, or other requirement designated critical.			
5	None Reported	N/A	N/A
Severity 2: Adversely affect the accomplishment of an essential capability and no work-around solution is known			
4	Business Logic	1*4=4	4*4=16
Severity 3: Adversely affect the accomplishment of an essential capability but a work-around solution is known			
3	Business Logic	3*3=9	2*3=6
Severity 4: Results in user/operator inconvenience but does not affect a required operational or mission essential capability			
2	Business Logic	7*2=14	15*2=30
	Field Validation	1*2=2	6*2=12
Severity 5: Any other issues.			
1	Business Logic	1*1=1	1*1=1
	Usability	5*1=5	7*1=7
	Naming Discrepancies	5*1=5	N/A
	Extra Fields	6*1=6	1*1=1
	Summary	**46**	**73**
MBTs severity score was ∼60% higher than manuals (73/46=1.59)			

The MBT tester focused mostly on functional issues. Adding a check for field names could have been included in the model but this type of testing does not fit well into the FSM approach because it focuses on the expected behavior of the SUT. Creating a simple Selenium script to check this would have been the best way to detect such ONBs.

Also, since the models for MBT were focused on functionality issues (Business Logic) we expected that MBT would perform well in this category. The same is true for field validation; MBT models typically test many possible types of input values both nominal and off-nominal (e.g. long inputs, inputs with special characters, no inputs etc.). MBT is very well-suited for such testing because a large number of test cases are needed to cover all necessary scenarios. It would take a human tester much more time to achieve the same type of testing. In addition, a manual tester might make mistakes in entering the test data thereby invalidating the test case by accident.

The MBT tester missed 5 of the business logic ONBs that were reported by the manual tester. Since our assumption was that MBT would perform better than manual for these types of ONBs, we investigated why they were missed. As a summary: two ONBs were covered by the model and could have been detected if more test cases had been generated. Three ONBs were not modeled and could not have been detected.

ONBs 1 and 2: The first ONB was found by the manual tester in two places of the SUT where the user has to enter data into a list of fixed size (for simplicity, the size is 4 in the example, but in the system the list size was 5 and 20, see Figure

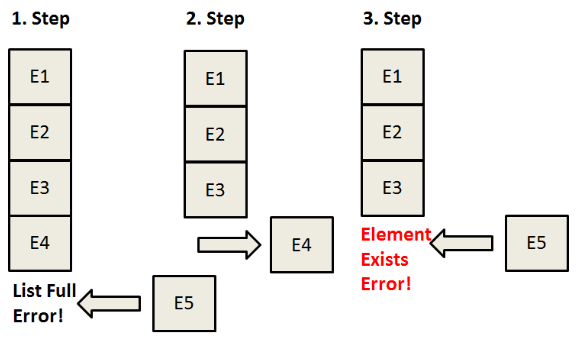


Figure 3.4: Example of an issue missed by MBT

3.4). To trigger the ONB, the tester has to perform three steps. Step 1: fill up the list with four elements E1E4 and then try to add a fifth element E5. At this point, the SUT correctly complains that the list is already full. In Step 2, the tester can remove element E4 from the list, thus the SUT still behaves as expected. Step 3: If the tester again tries to add element E5, the SUT will incorrectly complain that the element is already in the list even though this is clearly not the case. However, if in step 3 the tester instead tries to add an element that is different from E5, for example E6 or E4, then the SUT behaves correctly and accepts the element as expected. This incorrect behavior does not occur unless the tester first tries to add an element when the list is full.

MBT missed that ONB because of the way the input data was chosen by the testing framework. The SUT has several options for the user to choose from so the MBT framework randomly picked one that was not already in the list and tried to add it. Since it picked a random element every time it performed an action in the list, the combination of trying to add E5, removing E4, and trying to add E5 again never occurred because the chances for exactly this sequence to occur are low.

Thus the model and the testing framework contained the possibility for creating a test case that would catch this ONB, but the probabilities for it to get generated are low. We interviewed the manual tester and asked why and how the tester came up with the input data for the test case and the tester mentioned that a similar error was found in another system. Since the manual tester had experience with testing similar systems at GNSI, a very specific sequence was tested. This defect is actually similar to many of the business logic defects that MBT did catch. Thus, if the manual tester had used MBT, and used previous testing experience from GNSI systems to create the model, this and many other sequences could have been covered by the model and would potentially have found many more ONBs.

ONBs 3 and 4: Two ONBs were found in the zip code field. The zip code field did not handle foreign zip codes correctly and the combination of choosing the value other for the state field in combination with any zip code caused an error message. MBT missed both of these ONBs since the tester has knowledge about the MBT process but is not an expert on the specific domain that was tested in this case.

ONB 5: The last ONB was related to a session problem and occurred in the following sequence: 1. The tester opens an existing record for editing. 2. The tester deletes a required value and clicks on save, which correctly returns an error message. 3. The tester clicks on cancel and the SUT responds correctly by displaying the detailed view page. 4. When the tester clicks on edit again, the required fields value is blank, which is incorrect. Thus, even though the changes were not saved, the SUT behaves as if they were. However, the state was only local to this page and not saved to the database. Thus, when the record was loaded again, the correct values were displayed. MBT did not catch this ONB since the model did not contain a check for the existing data when invalid new data was entered.

3.3.4 Results and Observations: Effort

3.3.5 Effort Results

Each tester reported effort according to the following tasks.

- **System Understanding/Exploration:** Analysis of the documentation consisting of use cases and requirements. This involves inquiries to the developers as well as exploration of the SUT. Discovered ONBs were reported.
- **Modeling:** Creation of test models based on information gained in the system understanding and exploration step. This task is only necessary for MBT.
- **Implementing Test Infrastructure:** Creation of test infrastructure allowing automatic execution of generated test cases. This task is only necessary for MBT.
- **Test Case Development:** Writing (or updating for V2) test cases based on the requirement elicitation information. This task is only necessary in manual testing.
- **Test Execution and ONBs Analysis:** Execution and analysis of the result of the test case. This is reported as one activity since the two tasks are entwined in manual testing and cannot be reported separately. The manual

Table 3.9: Initial effort in person hours sorted by category

Task	**Manual**	**MBT**
System Understanding/Exploration	16 hrs.	16.5 hrs.
Modeling	N/A	24 hrs.
Implementing Test Infrastructure	N/A	87 hrs.
Test Case Development	16 hrs.	N/A
Total	32 hrs.	127.5 hrs.

tester analyses and confirms the validity of the detected ONB while executing the test cases and then documents how to reproduce them.

- **ONB Analysis:** Analysis of the results from the automated execution (failed and passed test cases) to confirm the validity of the ONB and to document how to reproduce it. This task is the MBT version of the manual task above since test execution takes no significant effort.

Since the workflows of the two approaches are different, the categories are also different, thus preventing a direct comparison. We therefore grouped the tasks into three high-level tasks: 1) *Initial effort* including all activities until testing can actually start, 2) *Effort for testing V1* and 3) *Effort for testing V2.*

Table 3.9 shows the initial effort that the two teams spent to get ready for the testing phases. Both teams spent around 16 hours on system understanding and exploration, which includes analysis of the requirements, questions to the develop-

Table 3.10: Effort for testing V1 in person hours

Task	**Manual**	**MBT**
Test Execution and ONB Analysis	26 hrs.	N/A
ONB Analysis	N/A	4 hrs.
Total	26 hrs.	4 hrs.

ment team and system exploration. Modeling was only done by the MBT tester who spent 24 hours creating the testing models of the SUT. Implementing the test infrastructure is also a task that was not carried out by the manual tester. It took the MBT tester 87 hours to implement the necessary functionality to execute the generated test cases automatically. Test Case Development was only carried out by the manual tester since the test cases for MBT are derived automatically from the testing models. Creating the manual test cases took 16 hours. The total initial effort for MBT was therefore 127.5 hours and for the manual tester it was about one fourth of that effort, i.e. 32 hours.

The two teams developed the test cases for V1 of the SUT so there was no effort necessary to adapt the test cases to this version. Testing V1 took the MBT tester a total of 4 hours and the manual tester 26 hours (see Table 3.10). The manual testing task consists of test execution and issue analysis. Since these two sub-tasks are highly entwined it is impossible to log how much time was spent on each one. At the same time, the MBT tester only had to do the issue analysis since the actual test execution was done by the automated test execution framework, which just takes

Table 3.11: Effort for testing V2 in person hours

Task	**Manual**	**MBT**
Adapting test cases for new version	0 hrs.	6 hrs.
Test Execution and ONB Analysis	7 hrs.	N/A
ONB Analysis	N/A	2 hrs.
Total	7 hrs.	8 hrs.

Table 3.12: Overall effort

Task	**Manual**	**MBT**
Overall Effort	65 hrs.	139.5 hrs.

computer time (three hours) and no human effort.

For V2, the manual tester and the MBT tester applied the same test cases that were developed for V1 on a modified version of the SUT (V2). In this new version, the GUI of the SUT was changed drastically. Adapting the test cases for these changes into account took 6 hours for the MBT tester while the manual test cases did not have to be adapted. The test execution and ONB analysis took the manual tester 7 hours. The ONB analysis with automated test execution took the MBT tester 2 hours. The total effort for testing V2 was 7 hours for manual testing and 8 hours for MBT, see Table 3.11.

In order to summarize the effort analysis: the overall effort of all tasks was 139.5 hours for MBT and less than half the effort (65 hours, 47%) for manual testing, see Table 3.12.

3.3.6 Effort Discussion

The two testers used about the same effort to analyze the requirements. This is expected since the tasks in this phase were the same. Most of the effort for the MBT approach was spent on implementing the testing infrastructure. About half of that effort was spent because of issues that the MBT tester had to resolve with the test execution framework. These hours included learning additional technologies, which are one-time efforts and could be reused in similar systems with a fraction of the effort.

The first issue that the tester encountered, while implementing the testing framework, was a problem with automatically validating the correctness the system response by inspecting a complicated table structure. Selenium and Java were used to create a small program to extract the structure from the webpage. The structure was then queried by the test case to judge the correctness of certain data entries. This testing solution, which can be reused in other testing tasks, resulted in five Business Logic issues, but took much effort to implement.

The second issue was due to the fact that selenium could not be used to test all parts of the webpage. For entering the path to a pdf file the system prompts the user with a native system window to choose a file from the file system. Selenium cannot handle native windows. In order to drive the test execution, the MBT tester had to integrate another technology into the test execution framework.

Implementing the solutions for these issues took about 40 hours and was carried out by the student intern who supported the tester. Aside from these issues, the implementation of the test execution framework was relatively simple, but due to the large number of elements on the data entry page, the edit page, and the approval page it still took around another 40 hours to implement and debug.

The changes in the SUT between the two tested versions did not affect the functionality, only the GUI. The manual tester did not have to change the test cases to adapt them to the new version since they were on a higher level of abstraction than the changes in the SUT. For example the manual test case just mentions navigating to a certain page and does not contain detailed instructions about which buttons and links the tester has to click on to get there. The same was true for the models that were developed for MBT; they were abstract enough and did not have to be changed. What had to be changed was the test execution framework and the mapping table.

The test execution framework uses identifiers or locators to select or locate elements on the web page. Most of these changed in V2 and had to be replaced or updated. In addition most of the navigation through the SUT (i.e. how to get to a certain page) also changed, but these changes did not affect the model because these details were in the testing framework stored outside of the models. The impact of this can be seen in Table 11, which shows the effort data of adapting the testing framework so it could test the new version of the web page. This shows that humans can adopt much better to changes on GUIs than automated frameworks.

The data shows that the initial effort for manual testing was about 25% of the MBT effort. This is a lower number than in the effort model presented by Utting et al. [22]. In their model the initial investment for MBT was only slightly higher than for manual testing (60 hours vs. 50 hours). However it is difficult to compare those numbers directly since the characteristics of the SUT in their study lacks details. We just know that it is a theoretical model of a small application. But these characteristics can have a high impact on the necessary effort. From our experience with other systems, we have learned that implementing a testing framework for GUI-based systems often requires more effort than for an API-based system.

Our interviews and observations also indicate that there are overlaps between the two testing processes; mainly in the system exploration and requirements analysis. The data also shows that the two testers spent roughly the same amount of time on these activities.

3.3.6.1 ONB and Effort Discussion

Here we analyze effort and ONBs together. Table 3.13 shows the effort in hours per detected ONB for MBT and for the manual testing.

On average, the manual tester spent 2.24 hours per reported ONB whereas the MBT tester spent 3.86 hours. We added a third comparison point named MBT* that excludes the one-time cost to develop the solutions in the testing framework discussed earlier in this section. This comparison shows a slight lead for the manual testing task. However, this number is expected to shift in favor of the MBT process

Table 3.13: Effort per ONB overview

	Effort	**ONBs**	**Effort/ONB**
Manual	65 hrs.	29	2.24 hrs.
MBT	139.5 hrs.	36	3.86 hrs.
MBT*	139.5 (-40) hrs.	36	2.76 hrs.

Table 3.14: Effort per severity score

	Effort	**Sum of Severity Score**	**Effort/ Severity Score**
Manual	65 hrs.	46	1.41 hrs.
MBT	139.5 hrs.	73	1.91 hrs.
MBT*	139.5 (-40) hrs.	73	1.36 hrs.

if more iterations of the testing process are needed. This analysis assumes that each ONB is equally severe.

By taking the severity scores into account (Table 3.14), we can compare the value of the two approaches. Manuals severity score was 46 compared to MBTs 73. Thus, Manual testing required 1.41 hour per severity score while MBT required 1.91 hour. If the 40 hours special set up cost is deducted then MBT* required 1.36 hours per severity score.

This comparison still shows a slight lead for the manual testing task unless we deduct the special set up cost. In the case that yet another version of the same

Table 3.15: Test Execution and ONB Analysis Effort

	Effort	**ONB**	**Effort/ONB**	**Sum of Severity Score**	**Effort/Severity Score**
Manual	33 hrs.	29	1.14 hrs.	46	0.72 hrs.
MBT	12 hrs.	36	0.33 hrs.	73	0.16 hrs.
Factor	2.75 33/12	0.81 29/36	3.45 1.14/0.33	0.63 46/73	4.50 0.72/0.16

system needs to be tested, then we expect the manual effort to be higher than the MBT effort. This is because the manual tester must manually test the system again while the MBT tester can update the model and then automatically rerun the same test cases again without any significant effort.

We can speculate how much it would cost to test the next version of the same system. By dividing the total Test Execution and ONB Analysis Effort by the number of ONBs we get 1.14 hours/issue for manual and 0.33 hours/ONB for MBT. When we take severity into account, we get 0.72 hours/severity score for manual and 0.16 hours/severity score for MBT. Thus, once the testing infrastructure has been implemented and working correctly, we expect the testing and analysis effort to be reduced (by a factor 2.75). The effort per detected ONB can also be expected to be reduced (by a factor of 3.45), as well as the effort per severity score (by a factor 4.50). Based on these numbers, we can also speculate when MBT would start to pay off.

If we assume that the severity score will be constant over the next set of versions and that each version has a summary severity score of 48.5 (97 = union of 46 and 73 from Table 8)/2 that must be detected before testing stops, and that the six hours to update the model/testing framework (which is included in the MBT effort) applies also to future versions (i.e. 3 hours per version), then we get the following expression:

$$32hours + x * 48.5 * 0.72hours = 127.5 + x * 48.5 * 0.16hours$$

$$\frac{95.5}{0.56 * 48.5} = x- > 3.51versions$$

Of course, since we only have two data points (V1 and V2) we must be aware of this limitation. In addition, since in real life it is not known how many ONBs or what types of ONBs are present in the SUT, it is impossible to have a certain severity score as the stopping criterion. At the same time, these calculations allow us to reason about when a technology such as MBT might pay off. In this case, based on our assumptions and simplifications, it would have paid off if two more versions of the SUT had to be tested.

3.3.6.2 Effort to Analyze ONBs

By adding the MBT analysis effort and dividing by the number of detected ONBs we get a measure of the effort to analyze one MBT detected issue: 6 hours / 36 ONBs = 10 minutes per ONB. If we assume that more severe ONBs take longer to analyze (which is definitely true for many of the business logic ONBs as compared to usability, extra field, and naming discrepancy issues) and take severity into account, we get 6 hours / 73 severity scores = 4.93 minutes per severity score.

3.3.7 Lessons Learned with regards to MBT

We learned many lessons related to MBT. For example the MBT tester created two models: one large model that covered most of the SUTs functionality, and one small model for testing the different user roles. The models, as well as slices of them were used to generate test cases. A model slice is created by copying parts of an existing model into a new one to focus on a specific aspect (e.g. the data entry). While this allowed the MBT tester to create focused test cases, it also created a maintenance problem. It was difficult to synchronize the slices (copies) when they were updated. Instead the MBT tester should have divided the SUT into complex and non-complex parts and should have created smaller specialized models for the parts with a higher complexity and separate models or automation scripts for less complex parts.

An example of this was the data entry page of the system. It has many simple text fields that were grouped together in the model by topics (e.g. address) and for each topic area the model had transitions for the different boundary conditions (e.g. field length, special characters etc.). This made the model and the resulting test cases large. This level of fine-grained testing could have been more effectively handled via separate models that just test this aspect of the page, thus making the model and the resulting tests less complex. Other models should have focused on the more complex and repetitive aspects of the system.

Another lesson is related to the lengths of the test cases because long test cases can be difficult to analyze. The problem of a failing test case is to figure out which parts of the test case were responsible for its failure. Since the test cases are generated in a random fashion and the test execution is fully automated, the tester does not know what actions led up to the failure. The manual tester has an advantage here. The manual tester sees the response to each input and therefore can easier identify the cause of the issue. The MBT tester might have to retrace some of the steps in the test case before being able to identify the cause. Both the smaller and more specialized models and the advanced model slicing would have resulted in more, but shorter test cases that would have helped make the test analysis easier.

One of the points that the observer mentioned was that the test cases were disconnected from the model. That is, during the analysis of the test case results, the MBT tester did not use the model anymore. Asked about that fact, the MBT tester replied that it was difficult to go back and forth between the model and the test case because there was no direct link in the test case to the location in the Model. This fact can be overcome by adding information about the state and transitions into the generated tests.

Another lesson was that some features of the SUT would have been easier to model if the MBT tool would have supported extended finite state machines (EFSM). An EFSM is an extended version of a FSM that adds variables and guards. The variables can be used as counters for example and the guards can be used to open or close transitions if the variables have a specific value. The data entry form had a list where the user can add entries until a certain limit is reached. This kind of behavior can be modeled better in EFSMs than in FSMs.

An interesting observation was that there seem to be overlaps between the testing processes. Namely in the system exploration and requirements analysis were the testers spent nearly equal amounts of time on these activities. An interesting question would be, if the manual tester already built this knowledge into the test cases that he/she produced can their knowledge and insight be leveraged in the model and test infrastructure creation process which are the most effort heavy aspects of MBT.

3.3.8 Threats to Validity

This section is based on Wohlin et als [88] four classes: threats to internal, external, construction, and conclusion validity.

3.3.8.1 Threats to Internal Validity

Threats to internal validity are caused by factors that were not considered but might have influenced the results of the study.

The testers were instructed not to share information about their tasks and the testers did not communicate during their testing sessions. It is of course not possible to fully control this if the testers are not monitored at all times. Nevertheless, based on the differences of the detected ONBs we saw from both testers we do not believe there was any interchange between the two testers during the testing session. Thus there is a low risk that the test results were due to other factors than then ones described above.

There was however an accidental connection between the two testers during testing of V1. The MBT tester did not clean out data between the testing sessions properly. The data that was entered was not well formed and the SUT should not have accepted it as valid input. Thus, the MBT tester reported this as an input validation issue. Any attempts to further work with this data caused the system to crash. Since the data was left in the SUT, the manual tester used it in the testing process and reported the crashes as ONBs. However, because the crashes were symptoms of the invalid data they were excluded from the study. In order to avoid such interference while testing V2, the testers were instructed to remove data after testing. The risk here is that more symptom-based ONBs were incorrectly reported, however, the ONBs were analyzed from this point of view so this risk is low.

3.3.8.2 Threats to External Validity

Threats to external validity cover aspects that might influence the generalizability of the study. For example, are we likely to arrive at the same results if the study is replicated?

The SUT is web-based so we believe the results, conclusions and lessons learned in this study are valid for web-based systems as well as for similar GUI-based systems that are related to data entry, data searches, data reporting etc. for which automation technologies similar to Selenium exists. Manual as well as model-based testing is dependent on the person conducting it. However, we believe that someone with the same experience and background as the testers in this study would arrive at similar but not exactly the same results. Since the testers in this study were representative of many people in the IT-industry, we did not identify any significant threats to external validity except for natural variation between individuals.

3.3.8.3 Threats to Construct Validity

Threats to construction validity assess if the correct measurements were used in the study. We measured the effort in hours and counted and classified the number of issues. These are direct measures that are not derived and therefore good indicators for the efficiency and effectiveness of the two approaches. Assigning severity scores to the detected issues, is however, a subjective exercise, thus there is a slight risk that someone else would assign these scores differently.

3.3.8.4 Threats to Conclusion Validity

Threats to conclusion validity cover issues that affect the ability to draw the right conclusions from an empirical study. We used professional workers instead of students and let them work on the tasks for several weeks instead of just during a few hours. Professional workers know the process better than students and do not need training. They are already experts in their field which will most likely yield different insights than those one can get from using students. However, using professional workers is very costly and can interrupt the regular work schedule. We therefore had to limit the number of participants to only two testers and to two test iterations. Thus the collected data is relevant, but limited, which might have impacted the conclusions. Only more studies can show the validity of the conclusions.

3.3.9 Conclusion

We compared the effectiveness (ONBs found) and efficiency (effort spent) of completely manual testing as performed by a tester at a software company with model-based testing as performed by a tester at a software research center.

The results show that MBT detected more ONBs with higher severity scores with less testing and analysis effort than manual testing. However, it required more initial effort, which eventually would pay off if several versions of the same system would be tested. The initial cost would also be offset by applying the same approach to other similar systems by reusing the same infrastructure.

In addition, MBT is not equally well-suited for all types of testing tasks. MBT is good at detecting functional issues and ensuring that corner cases of data inputs are considered, but it is not equally well-suited for the verification of the layout of a graphical user interface or to check that text labels on the GUI matches text labels in the specification. Instead, it is often easier to let manual testers verify these aspects of the system. Furthermore, the MBT development process still has a manual component (in addition to creating the model) when it comes to the analysis and exploration of the system. Thus, some manual work is still necessary.

Based on the experiences of this study we identified lessons learned that can improve the MBT approach. We also started implementing and evaluating the improvement ideas for the MBT process and have already built some of them into our process.

3.4 Lessons Learned with Regards to Testing ONBs

This section discusses the lessons learned with respect to applying MBT to test ONB.

3.4.1 The MBT process helps to identify ONBs

The MBT process is very structured. Testers analyze the system behavior by studying the specifications or by exploratory usage of the system and encode this information together with the testers intention into the testing model. The testing models in our case studies showed clearly how the system should be used and where very helpful to talk to our industrial partners when we wanted to clarify questions about system behaviors.

In the models you can clearly see in what state the system has to be in order for a command to succeed (E.g. publishing a message before being able to retrieve it). The same action could then be added in into other parts of the model as an ONB (e.g. retrieving a message before it has been sent, or trying to retrieve the message again).

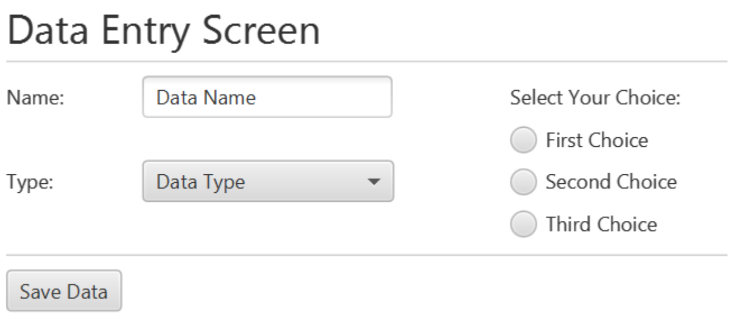


Figure 3.5: Simplified data entry form from the GNSI case study SUT

3.4.2 Two types of triggers for ONBs

The case studies show that there are two types of triggers for ONBs in testing models:

The first trigger type for ONBs is not present in the specifications and they have to be added to the model explicitly to cover them. The triggers are based on invalid or unknown sequences of events or data. Often times the requirements of a system do not mention, or only hint at how the system shall behave if the series of input sequences or the input data is invalid. The tester has to learn how to look for these triggers and add them to the model. However, as mentioned in the previous section. The structured nature of the process and the model helps to come up with potential off-nominal scenarios. An example from the Elexnet case study illustrates this issue.

Figure 3.5 shows a simplified data entry form of the SUT. The requirements state that the user has to enter a name and select one of the choices in the form and then press save to add the data to the system. The model-based tester considered, four off-nominal scenarios. Three are different inputs for the name field (no input,

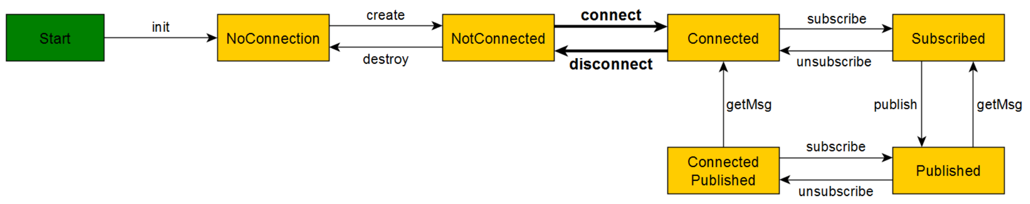


Figure 3.6: Simplified Nominal Model of GMSEC

long input, special characters as input) and one was an off-nominal scenario for the selection of the choices (the scenario was to selecting none of the choices). The requirements did not specify what should happen in case of the off-nominal scenarios. However, both the behaviors and their outcomes appear in the system. These extra paths through the system identified a critical issues in the Elexnet System. If no choice was selected in the system would then create one instance of the data for each non selected choice, which in the Elexnet system could be more than a hundred. This slowed down and even froze the system for several minutes afterwards until the data could be deleted.

The second type triggers for ONBs are inherent to the nominal model. The model can generate scenarios based on valid sequences of events that are not anticipated in the specifications and that can identify ONBs. Figure 3.6 shows a simplified nominal model of the GMSEC system to illustrate the ONB. The model describes the creation process of a connection for the GMSEC bus and contains some actions that the connection can perform once it is connect. The connection can then be disconnected and destroyed. This corresponds to the general workflow of a GMSEC app.

However, if the connect function is immediately followed by the disconnect

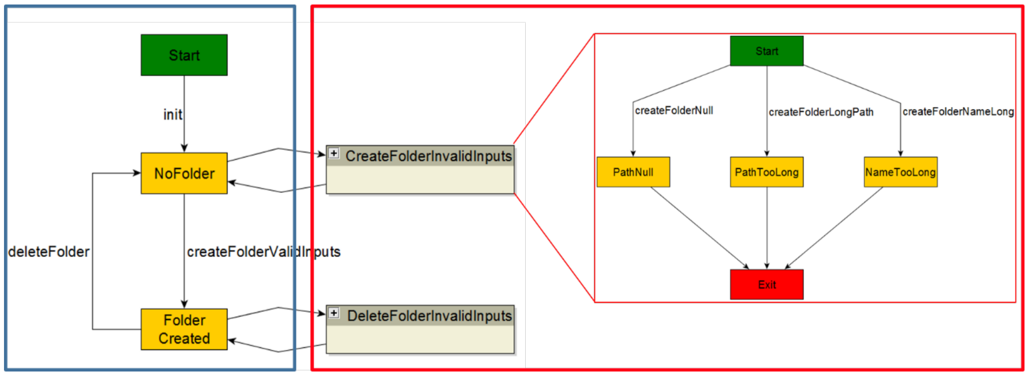


Figure 3.7: Example model that contains ONB. The nominal part can bee seen on the left side of the of the model. The off-nominal parts are in the submodels (grey states) an example of a submodel of with ONB can be seen on the right.

function, several times in a row an ONB is occurs. After some iterations the connect function call returned an error message and no connection could be established. Test generation approaches based on the structure of the model do not always identify these ONBs, in particular in larger models, and the tester has to manually guide the test generation process. This was the reason why 2 similar issues in the Elexnet system where missed by the MBT tester.

3.4.3 Off-Nominal Inputs dominate the models

In all three case studies, a large percentage of the models was dedicated for potential ONBs of the systems. This increased the model complexity, which directly affects the readability of the model and the effort for all subsequent tasks in the MBT process (e.g. creation of the test infrastructure, test generation/execution, issue analysis).

Figure 3.7 shows a simplified example of the model from the OSAL case study

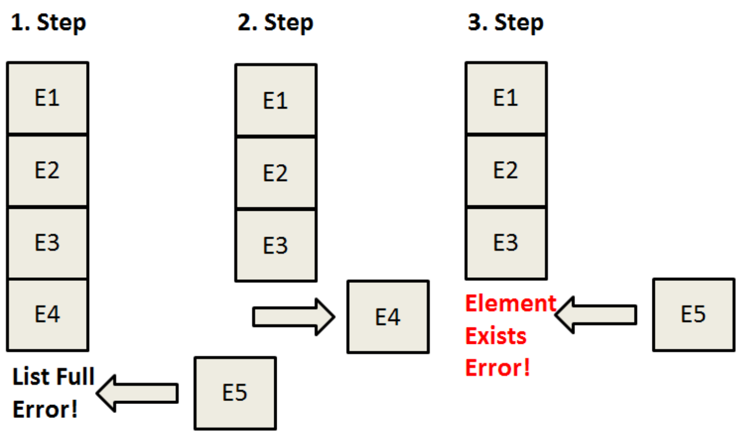


Figure 3.8: Visualization of an issue missed by MBT in the Elexnet Case Study. To trigger the issue the user had to populate the list in a specific order. The manual tester found the issue because of a previous encounter of a similar issue in a different system.

that illustrates this problem. The OSAL system is modeled as a hierarchical FSM and the grey states with the plus sign on top contain sub-models. The example model represents the creation and removal of a folder. The sub-models in the example contained the ONB of the system. As you can see just the off-nominal scenarios of creating a folder with invalid inputs require more space than the whole nominal part of the model. Furthermore, this model does not even consider invalid sequences of actions (e.g. trying to create a folder without creation of a file system first).

3.4.4 Handling Sequences and Data in FSM Models

In order to create good test cases, the generation algorithm has to consider both combinations of sequences and combinations of data. An example from the Elexnet case study is used to demonstrate this.

The ONB depicted in Figure 3.8 was identified in two places of the Elexnet system by the manual tester but was missed by the MBT tester. The user of the Elexnet system has to enter data into a list of fixed size (for simplicity, the size is 4 in the example, but in the system the list size was 5 and 20, see Figure 24). To trigger the issue, the tester has to perform three steps. Step 1: fill up the list with four elements E1E4 and then try to add a fifth element E5. At this point, the SUT correctly complains that the list is already full. In Step 2, the tester can remove element E4 from the list, thus the SUT still behaves as expected. Step 3: If the tester again tries to add element E5, the SUT will incorrectly complain that the element is already in the list even though this is clearly not the case. However, if in step 3 the tester instead tries to add an element that is different from E5, for example E6 or E4, then the SUT behaves correctly and accepts the element as expected. This incorrect behavior does not occur unless the tester first tries to add an element when the list is full.

MBT missed that ONB because of the way the input data was chosen by the testing framework. The SUT has several options for the user to choose from so the MBT framework randomly picked one that was not already in the list and tried to add it. Since it picked a random element every time it performed an action in the list, the combination of trying to add E5, removing E4, and trying to add E5 again never occurred because the chances for exactly this sequence to occur are low. Thus, the model and the testing framework contained the possibility for creating a test case that would catch this ONB, but the probabilities for it to get generated are low. The manual tester was interviewed and asked why and how the tester came up with the input data for the test case and the tester mentioned that a similar error was found in another system. Since the manual tester had experience with testing similar systems at GNSI, a very specific sequence was tested. This example shows that the way of handling sequences and data in the case studies is not optimally suited for testing these types of ONB using state machine based MBT.

Other MBT techniques that we evaluated [94] at Fraunhofer include Microsoft's SpecExplorer tool [42]. SpecExplorer takes an abstract code like description of the SUT and input values and automatically generates an FSM from them that covers all possible sequences and data that the description allows. However, the user has to describe the sequences of actions that are allowed in regular expression like language. For scalability reasons the sequences and the amount of data has to be chosen carefully otherwise the state space of the resulting FSM would become too large. For a system like Elexnet that has so many data input choices it would only be feasible with a lot of data abstraction, which would prohibit us from finding this issue again. However, we were able to apply it very successfully to another software bus that was developed by NASA [94].

3.4.5 The value of manual tests

In all our case studies we could see the value of manually created test cases clearly. In the OSAL and GMSEC study the existing tests were invaluable for creating the test infrastructure that is necessary to execute the generated tests. In the GMSEC study we also inspected the existing tests to determine how the system shall behave in scenarios that were not explained in the specifications. And in the Elexnet Case study we could see that the insight and experience of the manual tester helped to identify complex issues that MBT missed.

It would be advantageous to be able to automatically leverage existing manually created tests cases in the MBT process, by turning them into initial testing models, so that the information entailed in them can be reused in the more structured MBT process. Furthermore, by automatically creating testing models from existing tests one can automatically create the test infrastructure as well, thus supporting two of the tasks in the MBT process that take the most effort.

3.5 Conclusion

This chapter presents three MBT case studies on a variety of system types. The results show that MBT was successful in all three case studies. MBT could detect many ONBs even in already tested systems. Most of the ONBs identified with MBT were in the Bug/Defect category, which is not surprising since the testing models were based on the requirements and off-nominal inputs that could find the Surprise categories of ONBs have to be added manually. However, the structured nature of the modeling process often already helps to identify ONBs. The formal nature of the models helps to identify contradictory or vague requirements. Thus, the models could serve as additional documentation that we could often use successfully to communicate behaviors of the systems to the stakeholders and developers. This is an important aspect for the identification and documentation of ONBs, since the model helps to clear up and codify the intended and expected behaviors of the system. Furthermore, the structured test generation aspect of MBT makes sure that many behaviors of the system are executed, in particular corner cases which manual testing often misses.

However, the lessons learned also identified several limitations of the MTB approach. The model and test infrastructure creation process requires effort and skilled testers that also need to be able to program. The addition of ONBs to the models makes this an even harder problem, since it increases the complexity of the models greatly. Ideally one would like to minimize the impact of these factors. This would make the approach more accessible, so that people with less training can perform it on a similar level than someone more experienced.

The next chapter will introduce and evaluate an approach that we developed as part of this thesis. The approach automatically creates initial testing models and their associated test infrastructure from existing manually created test cases. This approach is aimed at leveraging the knowledge and skill of the manual tester and use the MBT process to magnify it and to identify additional ONBs that the manual tester missed.

Chapter 4: Model Generation from Test Cases

As we have shown in the previous chapter, MBT is a promising and versatile testing technology. Nevertheless, adoption of MBT technologies in industry is slow and many testing tasks are performed via manually created executable test cases (i.e. test programs such as JUnit). In order to adopt MBT, testers must learn how to construct models and use these models to generate test cases, which might be a hurdle. An interesting observation in our MBT evaluations is that the existing manually created test cases often provided invaluable insights for the manual creation of the testing models of the system and its associated testing infrastructure. In this chapter we develop and evaluate an approach [25] that allows the tester to automatically generate a model and its associated test infrastructure from a set of manually created test cases. The generated model is derived from the test cases, which are actions that the system can perform (e.g. a button click) and their expected outputs in form of assert statements (e.g. assert data entered). The resulting model is a Finite State Machine (FSM) model that can be employed with little or no manual changes to generate additional test cases for the SUT. We successfully applied the approach in a feasibility study to the NASA Data Access Toolkit (DAT), which is a web-based data management and access system. One compelling finding is that the test cases that were generated from the automatically generated models were able to detect ONBs that were not detected by the original set of manually created test cases. We present the findings from the case study and discuss best practices for incorporating model generation techniques into an existing testing process.

4.1 Overview

We developed two software tools related to MBT: one tool that allows the tester to generate models from Selenese test cases (The Fraunhofer Model Generation Tool), and one tool that generates test cases from such a model (the Fraunhofer MBTCG Tool). The MBTCG tool provides a GUI to the Graphwalker Model-Based Testing tool. The MBTCG tool was developed to support the user and it incorporates several of the lessons we learned in chapter 3. It adds visual debugging capabilities that help the user to identify issues in the models and adds tracing information into the resulting test cases to better relate the generated tests back to the model.

The models can be visualized and edited using the Yed graph editor [98]. In this chapter we will describe the algorithm that transforms test cases to FSM models. We will also describe the workflow of the overall approach: 1) creating the manual test cases, 2) automatically creating models from those test cases, 3) automatically generate test cases from the models, and 4) automatically executing large sets of generated test cases.

4.2 Background: Selenium

Selenium [99] is a browser automation tool that is often used as a testing framework for web applications. Selenium IDE is a Firefox [100] extension that allows recording, execution and editing of selenium scripts. Selenium scripts are stored in the XML-based Selenese language. Selenium scripts contain high level actions such as clicking on an element present on the web page, or asserting the existence of an element on the web page.

Selenium scripts use Java Script to navigate and manipulate the Document Object Model (DOM) of the web application. A Selenium script can be used to simulate how a user uses the system to carry out a certain task. Through the use of various assertions, the script can automatically determine if the expected output is provided by the SUT. Thus such a script is a test case that can be used to automate testing of the GUI of a web-based system.

A Selenese command is a triple: <command, target, arguments >. The first element contains the name of the command (e.g. type), the second element contains the target of the command (e.g. an input field), and the last element contains the arguments for the command (e.g. the text that should be entered into the input field). Java script commands can be added to the test scripts, e.g. to create random names for input fields.

We divide the Selenese commands into two different categories. Assert commands and action commands. Asserts are all Selenese commands that verify certain aspects of the SUT (E.g. assert text present, assert element present), whereas actions are all commands that interact with the SUT (e.g. click a button, enter text).

4.3 Model Generation Algorithm

The algorithm transforms a set of Selenese test cases into a FSM testing model by employing a set of heuristic transformation rules, based on observations of our manually created testing models and test cases. Our FSM testing model is a quintuple (Σ, S, s_0, δ, F), where:

- Σ is the input alphabet. The input alphabet is determined by all actions in the test cases. Two Selenese actions are considered equal if their command, target and arguments are equal.
- S is a finite, non-empty set of states. The states are determined by the assert commands in the test cases. Two states are considered equal if their corresponding assert commands are equal (same ¡command, target, arguments¿). The algorithm adds helper states in case of two or more consecutive actions (e.g. if a click is immediately followed by a enter text without any assert commands in between). Helper states are identified by the command of the previous action and previous state.
- s_0 is the initial state of the FSM and an element of S. The label of the initial state is Start.
- δ is a state-transition function: $\delta : S \times \Sigma \to S$

- F is a (possibly empty) set of final states. In the case of our testing models final states are states that do not have any outgoing transitions.

We created a set of rules that transform selenium commands into transitions and states. The rules determine when to merge states and when to add helper states in case of two or more consecutive actions . The transformation rules are encoded as follows (see Figure 4.1).

The merging approach is implemented as a two-pass algorithm. In the first pass (lines 1 to 6) it traverses each test case and merges consecutive assert commands together (line 4). In case of consecutive action commands helper states are interleaved into the test case (line 6).

In the second pass the state machine is constructed from the modified test cases. The algorithm starts by creating the state machine and the start state s_0 (lines 7-8) and then iterates through each test case in the test suite. It then resets the *lastVisited* helper variable to the start state (line 10) and then iterates through the actions of the modified test case. This way it can produce models of single test cases as well as models of several test cases that are merged into one model.

In lines 12-13, the algorithm checks if the action is an assert-action (future state) and if that state already exists in the state machine. If it does not exists it will be created in the next line.

Lines 14-26 handle the actions (future transitions). It first checks if the action already exists as a transition in the current state (line 15). If it does it checks if the target of the transition is the same as the next state in the test case (line 16). If it is

```
//First Pass
for each TestCase in TestSuite
  for each command in TestCase
    if consecutive-asserts
      merge assert-commands
    if consecutive-actions
      insert helper state between actions

//Second Pass, on transformed test cases
create state-machine
create state $s_0$
for each TestCase in TestSuite
  lastVisited = $s_0$ //always start in start state
  for each command in TestCase
    if command is assert and not in state machine
      create state (command)
    if command is action
      if command already exists in lastVisited state
        if existing.target equals command.target
          lastVisited = existing.target
          continue
        else
          Error: Indication of possible non-determinism
        continue
      else
        create transition(command)
        transition.source = lastVisited
        transition.target = State(nextAssertAction)
        lastVisited = transition.target
```

Figure 4.1: Pseudo Code of the two pass model generation algorithm

then the *lastVisited* state will be set to this next state and the algorithm continues with the next action in the test case (lines 17-18).

If the target state of the existing transaction and the current test case action are not the same the algorithm will throw an error message to the user (line 20). In this case the approach would produce a non-deterministic state machine. This is an indication of an error in the test cases that can appear through manual modification of the recorded selenium test case. This case has been introduced into the algorithm as a sanity check for the test cases.

In case that no correspondent transition exist in the current state, a new transition will be created from the current action and connected to the *lastVisited* state and the next state in the test case. The *lastVisited* state will be modified to reflect the new state.

4.4 Workflow Overview

In this section we will describe the workflow of our approach, from the creation of manual test cases to model generation to test case generation. Instead of manually creating a model, which is the common manner to conduct MBT, the new approach starts with constructions of example system usages. That is, the tester creates a set of Selenese test cases by using the record feature in the Selenium IDE. Each test case can be replayed in the Selenium IDE and potential defects in the test case can be addressed. Then the tester automatically generates a model from these test cases, which is used to generate more test cases. The workflow is supported by the two tools described above. The tools manage the model generation, model creation and test generation. The workflow of the approach is as follows:

1. The tester analyzes the features of the website and decides which test cases to create. The division is either done by web-page or by use-case or a mix of both.
2. The tester records a set of test cases for each feature using the Selenium IDE.
3. Once the tester debugged the test cases, the tester uses the tool to generate a model for each feature.
4. The tester visually inspects the model and fixes issues, or adds missing behaviors.
5. The tester generates test cases from each model.
6. The tester executes the test suite and analyzes the results of the execution. Failed test cases are especially analyzed.

After finishing the separate features of the system, the tester now combines all test cases into a larger model of the system by repeating steps 3-5 for all test cases to generate system level test cases.

The tester can use different test generation strategies and block out parts of the model in order to guide the test generation into specific parts of the model. This can be useful for stress testing the system by creating certain objects over and over. E.g. in the example the tester could block off the *set template type report* transition in order to make sure that only templates of the *type plot* are created.

4.5 Case Study

The tester applied the approach described in the previous section to test the features of the DAT web-interface which consist of the web pages *Request Data*, *Manage Templates* and *Manage Repository*. This section will describe the features that we tested, the manually created test cases and the models generated from these test cases. Furthermore we will provide effort data for applying the approach.

4.6 System Under Test

The Data Access Toolkit (DAT) is an archive, access and analysis system for NASA mission data. DAT provides an advanced query interface for mission analysts, mission managers, and the flight operation team to search and mine the available data. Users can query the system using a web-based query interface based on representational state transfer (REST) or through a web-based graphical user interface (GUI) to query the underlying PostgreSQL database, which stores metadata.

It is important to mention that the version of the DAT system that we tested has relatively high quality both due the fact that it has been around for some time (the first build was produced in 2012) and because of the DAT teams testing efforts. The testing Fraunhofer conducted as a basis for this paper was regression testing and therefore a limited number of detected ONB are expected. Specifically we discuss how a Fraunhofer tester tested DAT through the GUI. The features that were tested are: *Request Data*, *Manage Repository* and *Manage Templates*. The tester used a set of DAT use cases provided by the DAT team to understand the workflow and planned the manual test cases accordingly.

The *Request Data* page is the data access page of DAT. This page allows the user to insert values into input fields that together form a query for data. When this query is submitted, the DAT system responds with the resulting data. The request data page implements a rich query language and offers several options and components for formulating queries. Testing that all possible request queries return correct data is therefore a daunting task that requires a large set of test cases.

The *Manage Repository* page allows the user to navigate a tree-like structure of the DAT repository. The tree-like structure of DAT consists of Repositories, Missions, Namespaces and Archives listed in the highest to lowest in the hierarchy. Thus, an instance of DAT can have many repositories where each repository can have several missions. Each mission can have several namespaces, and each namespace can have several archives.

Using the manage repository feature, repositories, missions, namespaces and archives can be created, edited and deleted. The data itself is stored in archives and can be configured to handle many different data types.

The *Manage Templates* page manages the templates of the system. Templates are used when requesting data and specify how the user wants the output to be reported and formatted. On this page the user can create, edit and delete templates. A template has a name, type, folder and a body encoded as XML. The name is the name of the template, the type declares how the data will be represented, a report or plot. The folder says what folder the templates should be saved in and the body is the declaration of the template in XML format.

It should be mentioned that the DAT team uses an agile software development approach. This development approach requires a testing approach that supports frequent changes and delivers testing results within short time. Therefore it was important to develop a testing approach that supports such short turn arounds.

4.6.1 The Tester

A student intern at Fraunhofer created the test cases manually, generated the model, generated test cases, executed the test cases and analyzed the test results. Before starting his internship, he had no background in Selenium IDE or Selenese, or MBT. As part of this internship he had about two months of experience with modeling and model-based testing on other projects before starting this task.

4.6.2 An example of applying the approach

In this example we explain steps 1-5 from the overview in the previous section with a simplified example. This will help understanding the algorithm, the workflow and the benefits of the approach. As an example we chose the *Manage Template* feature from the DAT system. The full description and size of the test cases the corresponding models and generated tests for the feature can be found in the case study section. We use a simplified example due to size constraints. In the case of the DAT system dividing up the features was straight forward since each feature had its own web page and the features are relatively independent of each other.

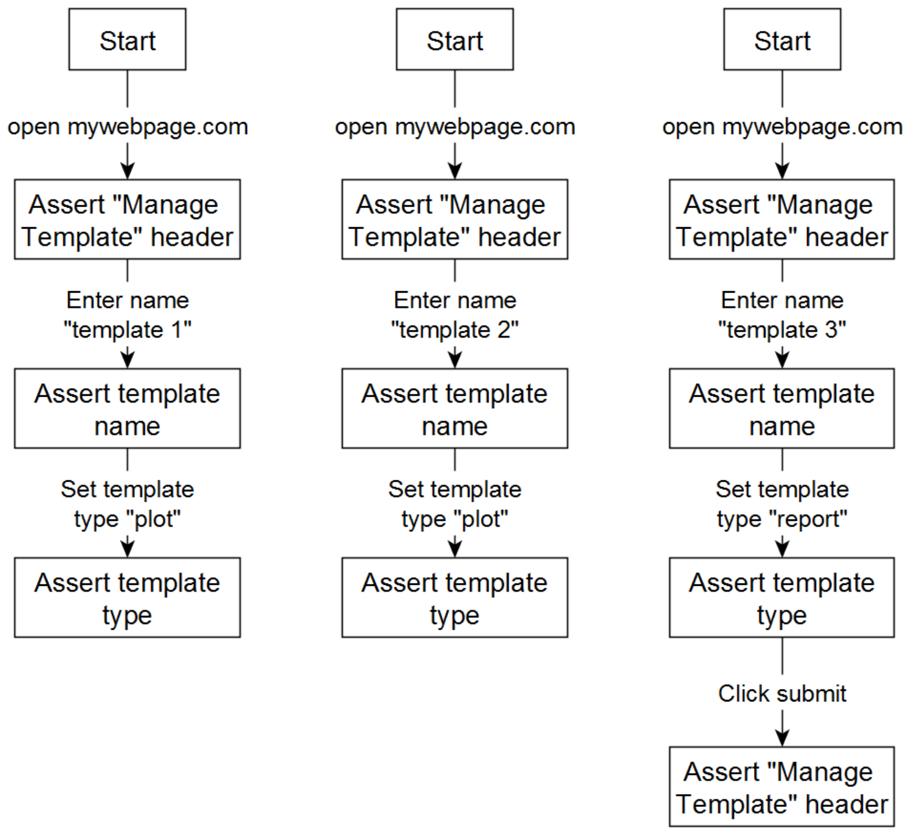


Figure 4.2: Three simplified test cases for the manage template feature

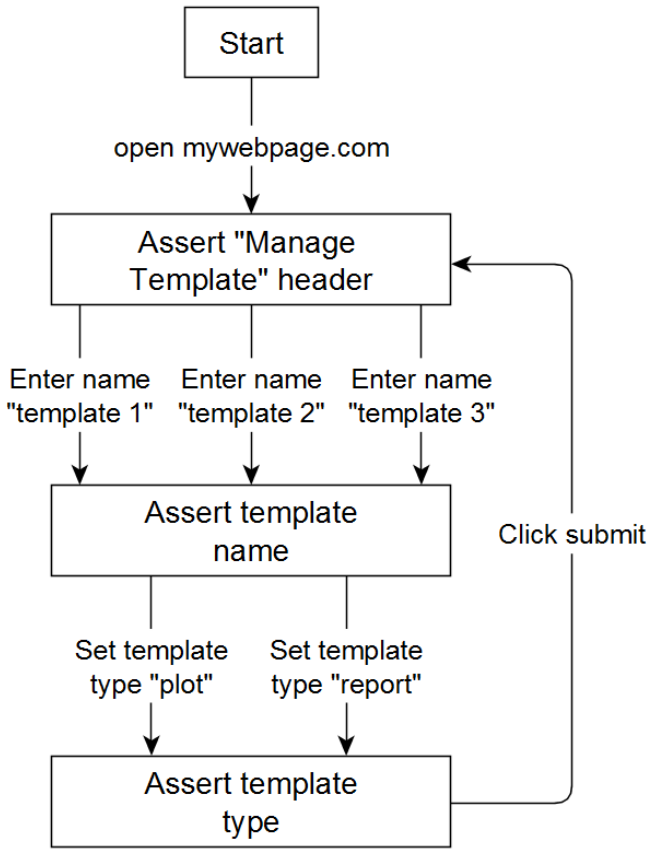


Figure 4.3: The Model of the combined simplified test cases

For this example, the tester recorded three test cases (see Figure 4.2) that test different variations of the possible inputs and options for creating a template. The tester replayed them using the Selenium IDE to detect and remove any potential issues. In some cases the tester also manually modified the test cases by adding commands that were not automatically inserted during the Selenium recording. For example, storing a value from the website in a variable that will be used in a later assertion on other pages. Adding such a command is supported by Selenium IDE and requires a right-click on the value to store in addition to the variable name.

When the manual test cases have been debugged, they are loaded into the model generation tool. In this example, the model in Figure 4.3 was created. The three test cases are merged in three merging points namely: the *Assert manage template header*, *Assert template name* and *Assert template type* actions. Thanks to the merging, the behaviors can now be interleaved to create 6 instead of three different inputs and can be repeated over and over again, thereby adding many more potential behaviors.

Before generating test cases from the generated model the tester first inspects it. Since the recording of test cases is a manual task there is the potential for errors. An example of an issue caused by such an error is a wrongly recorded assert command, which can manifest in unwanted paths or states in the model, or states that are not fully connected with the rest of the model. The model can be modified directly by adding, removing or modifying states and transitions. However, we have found that the best way is to instead edit the test case accordingly and regenerate the model. In this way, the model is never manually edited. The tester can also add, modify or remove test cases and in a matter of a few seconds regenerate the model.

The tester who was studied in the case study explained that after some practice he developed a habit to plan the test cases based on the testing goals and the model he wanted to generate to achieve those goals. Thus, he had an image in his head about which paths would be logical and what states should be created. He further explained that when he sees a path that he does not expect or a state that leads nowhere the tester knows that there is an issue with the model.

After the tester has verified the model he generates additional test cases from it and executes them against the system. He then inspects failing test cases. A test case can fail because of two reasons. Either the error is a true positive, which means that the test case failed because of a bug in the SUT, or the error is a false positive which means that the test case failed because of an issue in the model or in the manual test cases. The tester must inspect the generated test cases that failed and their execution in order to determine the nature of the issue. If the model or test case is incorrect, then the tester corrects them. If the test case failed because of an ONB in the system, then the tester documents the detected ONB including how to reproduce it, and reports it to the DAT development team. We will now describe how we tested the features of DAT using this approach.

4.6.3 The Request Data Feature

On the *Request Data* page the user inputs the parameters for a request to the DAT system. The page offers many different options and input fields. The user can enter a start date and time and then has the option to choose either an end date and time or to enter a duration (in days, hours, minutes, or seconds). Additionally the user can choose the different datasets to be searched. Testing this feature manually is difficult because of the many ways a query can be formulated and thus a large number of manual test cases would be needed to cover all possible options.

In the next step the user can add mnemonics. The mnemonics are the identifiers for the different sensory data that are stored in the system. In order to pick a mnemonic the user has to choose the mission to select mnemonics from. In the next step the user has to decide which properties (there are 14 different properties to select from) the user wants the system to return for each mnemonic. There is no known limit to the number of mnemonics that can be added.

To test this feature, the tester created 19 manual Selenese test cases with an average length of 15 instructions (assert commands and actions) and generated a model from them. The generated model has 55 states and 66 transitions. 32 of the states were based on assertions while the other 23 states were helper states introduced by the model generation algorithm. From this model, the tester created 100 test cases using a random traversal strategy. The generated tests had an average of 39 selenium commands. As indicated above, these test cases are executable and require no editing. The tester therefore immediately loaded the entire test suite into Selenium IDE, which executed the test cases automatically.

4.6.4 The Manage Templates Feature

The *Manage Templates* page allows the user to manage reporting and plotting templates. On this page the user can create, edit and delete templates. When creating new templates there are 2 input boxes and 2 dropdown menus (with two and 5 options to choose from in the dropdown menus respectively).

The tester created 5 manual test cases for the manage templates feature with an average length of 38 instructions. The generated model has 22 states and 27 transitions. 12 of the states are based on assertions and 10 are helper states. From the generated model the tester created 100 test cases using a random traversal strategy. The generated tests had an average of 35 selenium commands.

4.6.5 The Manage Repository Feature

On the *Manage Repositories* page the user can manage repositories, missions, namespaces and archives. Each of these items has a name and a description. The user can create, edit and delete repositories, missions, namespaces and archives.

The tester created 5 manual test cases for the Manage Repository feature that had an average of 27 instructions. The generated model had 28 states and 32 transitions. 6 of the states were based on asserts the other 22 states were helper states. For the test generation of this feature the tester used the blocking property of the MBT approach. This allowed the tester to focus the test generation on certain parts of the model without directly editing the model (i.e. deletion and rerouting of transitions was not necessary thanks to blocking). He did this by blocking certain transitions, which means that during test case generation, these transitions were not available. By blocking certain transitions the tester made sure that for example only archives could be created to study how the system handles the creation of a large number of archives.

The tester generated the following test cases from this model:

- A set of 10 test cases that cover the whole model and create repositories, missions, namespaces and archives. The test cases had an average lengths of 377 commands.

- Two test cases that only create one repository, but several missions, namespaces and archives. The test cases had an average of 1867 commands.

- One test case that creates one repository and one mission but several namespaces archives. The test case had a length of 1397 commands.
- One test case that creates one repository, one mission and one namespace but several archives. The test case had a length of 1864 commands.
- One test case that only create repositories. The test case had 1399 commands and created a large number of repositories.
- One test case that creates one repository and several missions. The test case had 1608 commands.
- One test case that creates one repository, one mission and several namespaces. The test case contained 1556 commands.

When the tester was creating the repository model he wanted to be able to create many entries in the same test case but was running into issues with naming because all names were static. The tester was able to find a way around that by using JavaScript code in the manual test case where he used the StoreEval to store a function in a variable. The tester stored a pseudo random JavaScript function that the test case always calls when creating a new template. The function creates a unique string throughout the run of the test case thus avoiding potential name conflicts. The function needed to be a pseudo random function because if a generated test case would fail, the tester needed to be able to run the exact same test case to be able to properly debug the issue.

4.6.6 Effort

The effort for the tester to learn how to create the test cases and transform them into models was about 8 hours. The effort for the tester to create the three suites of test cases that were input to model generation was 4 hours (manage repository), 6 hours (manage templates) and 12 hours (request data) depending on the model. The effort seems to be proportional to the complexity of the feature.

Executing 100 test cases with an average of 39 command each takes about 50 minutes. Each command takes about 0.77 seconds but this can be controlled via the speed setting in the Selenium IDE. The maximum speed is however often not feasible to use since the highest speeds often cause test cases to fail due to page loading issues.

It took about 12 hours to debug and run samples of the test cases. It takes about 16 hours to run all test cases, however, it should be noted that this is computer time so the effort for the tester is negligible.

The analysis of the failed test cases took the tester about 6 hours. This includes inspection of the failed tests and documenting them for the DAT team. In total, creating the test cases, generating models and test cases, running the test cases, and analyzing the failed test cases took about a week plus one day for learning (i.e. 48 hours).

4.6.7 Issues applying the approach

The tester observed two false positives in the generated test cases and investigated their causes. The first issue appeared in test cases that were generated from the Manage Repository model. The reason was that the manual test cases included a wait action after the creation of each repository, mission, namespace, or archive. This was done since the website reloads after the submit button is pressed and the tester has to wait since otherwise the next assert command would try to assert for elements that have not been loaded yet. The wait time in the manual test cases was set to 1500ms, but after creation of several entries in the system the reloading took more and more time and the test cases started failing. The tester increased the wait time in the manual test cases and recreated the model, which fixed this issue.

The second issue the tester encountered was caused by a dynamic identifier of an element that would change due to reloading the page. As mentioned in the previous section, Selenium tries to choose the best locator for an object but sometimes the best locator turns out to be a dynamic identifier, which changes between runs or by refresh of the web page. We also encountered such behavior in the Elexnet Case Study [26] and therefore the test cases fail at that point. To address this issue the tester had to change the type of locator of the recorded manual test. The tester had to choose a higher element in the DOM that was static and then used the xpath (i.e. a Selenium feature used to locate elements in an HTML document) to find the correct element in relation to the static element. In addition the tester also used the *contains* function and the *last* function in the xpath to navigate to the correct element.

4.6.8 Detected ONBs in the System Under Test

All three models generated test cases that detected ONB that were previously unknown by the DAT team. These ONB were not detected when the manually created test cases were executed. These ONB are:

1) Performance issue with representing large amount of data in the browser when requesting a table of data.

This ONB was detected on the *Request Data* page, which occurred when the tester was running test cases from the Request Data Model. The reason was that a generated test case had a large time range. The test cases included a large number of data points because of the large time range, which led to degraded performance of the GUI. Eventually, this made the test case fail. DAT (the browser actually) then returned a non-user friendly message that the script was *no longer responsive*. This is a violation of the documented behavior and thus in the Bug/Issue ONB category.

2) Data is retained in fields when creating multiple templates but not in other cases.

This ONB was detected when a test case attempted to assert a dropdown menu on the *Manage Template* page with test cases generated by Manage Template Model. The reason was that when a template is created and the type is set, the type was always identical to the previously created template. Upon further investigation the tester noticed that values were retained in an inconsistent way. The issue was that when the site was reloaded, the information of the last created template was not retained, but when two consecutive templates are created the information was retained. Thus, this behavior is inconsistent as to whether the data is supposed to be retained or not.

3) Templates are overwritten without any warning.

This ONB was detected on the *Manage Templates* page with tests generated from the Manage Template Model. When the test case asserts that the correct type of the template has been created, the type was sometimes incorrect. This was because only the XML of the template was being overwritten but not the type of the template. Let us assume there is an existing template with the name *tempTemplate*, with the type *report* and the xml *first xml*. If the test case creates another template with the same name *tempTemplate*, but with the type *plot* and the xml *second xml*, an inconsistency will occur. The reason is that this will lead to the template having the same name *tempTemplate*, the type *report* and the xml *second xml*. Using the updated *tempTemplate* will return an error due to this issue. This issue was noticed before the tester created a pseudo random function to create different named templates. This behavior was not documented and has negative impacts on the system and is therefore in the Bad Surprise ONB category.

4) Hidden view limit of archives.

This issue was detected on the *Manage Repository* page with test cases generated from the Manage Repository model where test case created a large number of archives. The issue is that there is an undisclosed limit on how many archives the system can display in the hierarchy. The limit turned out to be 100 visible archives.

If there are more than 100 archives in the system, they are not displayed and the hidden archives are impossible to reach. However, the user can still continue creating more archives although this limit has been reached. This issue was detected because the generated tests failed at exactly 100 archives. This is undocumented behavior with a negative impact and is thus also in the Bad Surprise ONB category.

4.7 Discussion

We have presented an approach where models are generated from test cases instead of being manually created. We will now discuss strengths and limitations of the approach as well as some other topics related to using it.

4.7.1 Strenghts of the Approach

- **Identification of ONBs:** The approach was able to identify ONBs that the existing manual tests missed.
- **Automated model generation:** Since the new approach can generate a model from existing test cases, it overcomes the hurdles related to creating models for MBT.
- **Automated mapping:** Since the model contains all information provided in the Selenese test case, the mapping problem is also reduced. The mapping problem means that for each transition and state in the model, the tester must assign executable statements in the form of selenium commands and assertions, which can be both tedious and error prone. The presented approach automat-

ically copies all executable commands into the model and thus minimizes the need for manual mapping.

- **Reduces the necessary skill level and facilitates learning MBT:** Since the new approach reduces the need for mapping, which typically is somewhat difficult as well as time consuming, the necessary skill level is reduced. Since the new approach is based on test cases, which testers know how to create, and these test cases represent examples of how the SUT it used (test cases), and since the tool shows what the corresponding model looks like, it is easier to understand and learn MBT.

- **Well prepared for regression testing of the next version of DAT:** Since the DAT team uses an agile development approach it is important that the testing approach can support quick turnarounds. With this approach we believe that this is the case and that a new DAT version of the GUI can be tested within one day since it takes 16 hours of unsupervised computer time to run all test cases. In case the GUI has changed significantly, we expect that creating new test cases and corresponding models will add between 8 and 16 hours to the effort.

4.7.2 Limitations of the Approach

Some limitations are due to the fact that we use MBT for test case generation, which has some well-known limitations. E.g. sometimes it was difficult to identify the root cause of an issue due to the fact that the generated test cases were long and are therefore difficult to comprehend. Also since there can be similarities in the generated test cases the same issue can often manifest itself in several test cases. E.g. 10 failing test cases can have the same root cause but this is usually only clear after they have all been inspected. Just because MBT can generate a large number of test cases does not mean that it is always helpful to do so. It is often better to guide the test generation using blocked commands and test different scenarios, which the tester did in a few instances.

Another limitation with the approach is that we have experienced that more advanced testing requires extended FSMs (EFSM), which allows guards and state variables, but the tool currently generates FSMs. The MBTCG tool supports EFSM, thus a manual step is required to turn the initial models into EFSMs as necessary.

4.7.3 Does this testing approach replace regular MBT?

We think this approach complements regular MBT because it provides a quick way to create initial models. In addition, it provides an excellent way of teaching MBT to testers who are unfamiliar with MBT. For example, the authors have already successfully used this approach in a graduate testing class to teach MBT to software engineering students.

4.7.4 Does practice matter?

Practice has a large influence on the approach, as expected, since the quality of the manually recorded test cases directly impacts the generated models. Furthermore, the tester mentioned that after some practice he already "knew" what the resulting model would look like when he recorded the test cases for it. This probably improved his ability to create test cases that would result in correct model and to inspect and debug the generated models. Thus practice made him more efficient.

4.7.5 Does this approach apply to non-web-based system?

In its current state the approach is tied to Selenium and therefore to testing of web-based systems. However, web-based systems are very similar to other GUI based systems and we see no reason why the approach could not be extended to other GUI based systems. The one restriction we place here is that for the approach to work well for testing GUIs of non-web-based system, a tool similar to Selenium IDE should be available for that GUI.

4.7.6 Comparing the models to manual MBT

We compared the generated models in this cases study to the manually created models in our previous case study [26]. One observation is that the generated models are on a lower level of abstraction. The manually created model in previous studies have no direct reference to the SUT and instead used abstract actions. The manually created model used for example an abstract command called submitdata to enter data into the system, whereas the automatically generated model is more concrete and for example has a reference to the id of the button directly in its label. The effect is that the generated models are slightly more difficult to understand due to the fact that they contain more details.

Another observation is that the labels in generated models are much longer than the labels in the manually created models. The reason is that the generated labels contain nearly all the information from the Selenese commands. The effect is that the readability of the generated labels is less than the manually created labels.

Another observed difference is the lack of sub models in the generated models. In the manually created models, the tester typically groups similar actions into sub models, which introduces a hierarchy to the model that is helpful for model comprehension. Currently the generated models are flat. The effect is that larger models are more difficult to navigate and to understand.

4.8 Threats to Validity

The threats to validity discussion is based on the model by Wohlin et al [88]. They define four classes of threats to validity, namely threats to internal, external, construct, and conclusion validity.

4.8.1 Threats to internal validity

Threats to internal validity are caused by factors that were not considered but might have influenced the results of the case study.

The results of this study show that the presented model generation approach was able to detect defects in the SUT. However, we have to consider the possibility that the tester who conducted the case study knew the ONBs before conducting the study and he could have therefore tailored the test cases and resulting models in a way that would make sure that the defects get detected. If that is the case the results could not be attributed to the approach but to the knowledge of the test engineer.

We believe this risk to be absent from our case study for the following reasons. Three of the identified defects were not known to the tester when this study started. The tester had encountered an ONB similar to the fourth detected ONB (retained template values). However, this ONB (or none of the other ONB) was not detected by the manually created test cases.

4.8.2 Threats to external validity

Threats to external validity are concerned with whether we can generalize the results outside the scope of our study.

We have to address different threats here. The first one is that the study was only performed by one tester and secondly the tester only tested one system. We are aware of this limitation. This is an initial feasibility study for our new approach and we plan to compare the results from this study with follow up studies that will have more testers, different systems and also different versions of systems.

Another threat is concerned with the knowledge of the tester. It is possible that another tester might not have been able to use the approach as effectively and therefore would have taken more time and/or might not have detected the same ONBs. And although it is easy to generate a large number of test cases with

MBTCG in some cases the tester has to block certain parts of the model in order to generate test cases for a specific scenario. This requires intuition and training. We addressed this threat by compiling a comprehensive tutorial for our test engineers that covers regular manual MBT (e.g. how to construct models for a system and how to generate effective test cases for different testing goals). We believe that this tutorial will lessen the variance between different testers. However, a natural variance between individuals is always given in these types of studies, no matter what kind of training is given to them.

The approach was designed specifically for Selenese and therefore only for web-based system. However, if there are similar record playback tools with a similar feature set for other types of GUI systems, we believe this approach would be applicable to them as well.

4.8.3 Threats to construct validity

Threats to construct validity assess if the correct measurements were used in the case study.

For this study we used direct measures such as the effort in terms of the number of hours spent on a certain task, the number of ONBs detected, and the size of the test cases and the generated models. We did not use derived or subjective measurements. The measures are therefore good indicators for comparison with other approaches and future case studies. The tester was given instructions to log the time that he worked on the different tasks so that the effort would be reasonably accurate.

4.8.4 Threats to conclusion validity

Threats to conclusion validity cover issues that affect the ability to draw the right conclusions from a case study.

Our conclusions are that the approach is feasible to use on industrial systems similar to NASA DAT in the described context. We currently do not see threats to that conclusion, mainly due to the nature of a case study like this one where no comparison to a control group is done. For such comparisons, controlled experiments are required.

4.9 Conclusion

This chapter presented an approach where some of the hurdles related to adopting MBT were addressed. In our cases studies (see chapter 3), we have observed that testers are not used to creating models, but they are used to creating executable test cases such as JUnit test programs and Selenium test scripts. The new approach avoids some of the hurdles related to manually creating models for testing by instead analyzing existing test cases and automatically generating models and their associated test infrastructure from those test cases.

There are many advantages with such an approach. For example, the tester can focus on creating test cases using tools like Selenium IDE where each test case can be automatically recorded by providing inputs to input fields, clicking on buttons, and adding assertions in proper places. Once the test case is created, the tester can ensure that it works properly by playing it again.

By generating a model from a set of such test cases, we showed how new test cases can be automatically generated. Since the new test cases are based on an MBT model that can repeat and interleave sequences they are more likely to identify ONBs. This could be seen in our case study were the tester found ONBs in the system that the original manual tests did not identify. It also reduces the problem that testers typically do not have the time to create enough test cases, with an approach like this they can create a small set of tests and then derive more from the model.

Since the new approach automatically copies all executable commands into the model, it contains all information provided in the Selenese test case, and thus the test infrastructure problem is also reduced. Maybe the most important points are that the case study shows that this approach is feasible since the case study was conducted in very reasonable effort, and that new ONBs were identified by the generated test cases ONBs that were not detected by the manually created test cases.

Chapter 5: Invariant Mining with Specstractor

This chapter reviews the Specstractor invariant-extraction framework and the associated tool chain that realizes the framework. The goal of Specstractor is to automatically extract system specifications in the form of system invariants from automatically generated tests using data mining. These automatically extracted invariants yield useful insight into the actual system behaviors. The invariants can be compared against existing specification artifacts of the system to identify and document ONBs in the form of violations of existing specifications or the identification of missing behaviors.

The Specstractor tool-chain consists out of the Extractor component, which is implemented in Java and C, and the web-based analysis tool Insight, which is implemented with the Angular 4 web-application framework. The Specstractor tool can be downloaded from `https://specstractor.sourceforge.io`. The website contains the instructions on how to install and use the Extraction tool[1], and it also contains a link to the Insight analysis tool.

The remainder of the chapter will present the Specstractor Framework, the tool chain and then discuss shortcomings of the framework that we will address in

[1]It should be noted that the execution of the Extraction Tool requires a trial license of Reactis and Reactis for C, which can be requested here: `www.reactive-systems.com`.

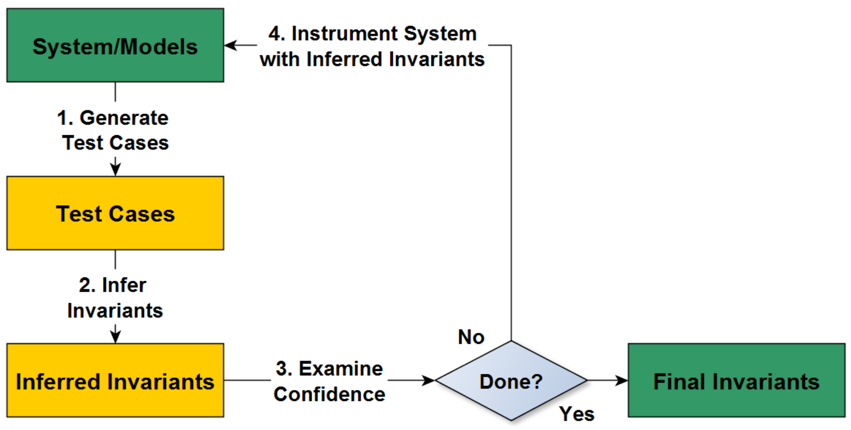


Figure 5.1: Overview of the specification extraction approach

the consecutive chapters.

5.1 Invariants via Testing and Data Mining

Our invariant-generation approach [20] combines automated testing with *association-rule mining* [101] to create invariants of a system. The general techniques follows an iterative *test* -> *infer* -> *instrument* -> *retest* cycle, and is depicted in Figure 5.1.

- **Generate Test Data.** Test data is automatically generated from system models using the Reactis®[2] automated coverage testing tool. Reactis actively strives to cover large portions of the systems behavior by targeting different structural coverage metrics, including decision coverage and modified condition/decision coverage (MC/DC), as well as Simulink-specific coverage metrics such as conditional-subsystem coverage. The input space that the test generator covers can be constrained (e.g. only generate speed values between 0 and 160 mph). In order to

[2]Reactis® is a registered trademark of Reactive Systems, Inc.

better address ONBs the test generator can be given free rein over nominal and off-nominal inputs to the system.

- **Infer Invariants.** Invariants are association rules, which take the form of implications whose left-hand sides (LHSs) refer to inputs and internal variables and whose right-hand sides (RHSs) are outputs of the system (e.g. $input1{=}1 \wedge internal1{=}1 \Rightarrow output1{=}1$). These outputs may themselves be new values for the internal state variables. The invariants are inferred from the test data that was generated in the previous step using a modified Frequent Pattern (FP)-Growth [35] algorithm from the Sequential Pattern Mining Framework (SPMF) [102][3].

- **Instrument System.** To validate the proposed invariants, the system models are instrumented with so-called *monitor models* that represent the proposed invariants. As their name suggests, these monitor models observe the underlying system for any violations of the associated invariants. Reactis provides support for automating this instrumentation processes. (In languages such as C/C++ this can be achieved via assert statements.)

- **Retest.** The purpose of the retesting phase is to check whether the inferred invariants can be falsified with additional testing. Since the invariants are attached to the models as observers, they are treated as additional coverage objectives by Reactis, which now actively tries to find counterexamples to the proposed invariants. In the process it either disproves invariants, or strengthens the confidence in them by creating additional test data that supports them. In addition to vali-

[3]The Magnum Opus tool used originally in [20] is no longer available.

dating existing invariants the additional test data can also uncover new behaviors which lead to invariants that were missed in previous iterations. The test data of all previous iterations is aggregated with the newly created test data and a new set of invariants is inferred in each iteration. If the monitor model detected a counterexample, then the corresponding invariant does not hold true for all test data and is automatically discarded by the data miner. Thus, iterating the approach leads to a more accurate set of invariants.

- **Terminating the Process.** The process terminates if there is no change in the set of invariants for N iterations, where N is a parameter that can be configured by the user. In our experience three iterations without a change in the invariants were sufficient.

The Extractor can be configured with different test generators, instrumenters and data miners; thus the general framework can be extended to support specification-extraction from a variety of system types and also to integrate new types of invariants. Currently Specstractor supports analysis of Simulink models and C code using the Reactis and Reactis for C test generators.

In order to adapt Specstractor for other systems the following requirements have to be met: *Automated test generation* technology needs to be available; *Observability* of the inputs, internal states, and output values that should be in invariants must be possible; *Monitoring* of the invariants during system execution, without changing system behavior, must be possible. It should be noted that some of the optimizations introduced in this thesis rely on the availability of data- and control flow information between the variables.

5.2 Analyzing Invariants in Specstractor: Insight

The results of the invariant mining process is a text file (see figure) with all the invariants and their associated meta data. Analyzing the text file manually is tedious, since there can be many invariants, and it is hard to compare them or search for invariants involving specific variables. We therefore developed the Insight tool, which helps the user to navigate and visualize sets of invariants. In addition to search and filtering options Insight also offers the user high-level views on the resulting invariants.

For systems with internal variables, Insight can construct state-machine views of these internal variables (left in Figure 5.2). All invariants that are involved with the chosen internal variables will be added as transitions to the state machine. This allows the user to reconstruct high-level operational view of the system, as well as slice the system behaviors according to different internal variables. To judge the impact of a specific variable or a value of a specific variable on the system we also created the impact score heat map (right in Figure 5.2). It shows how often each input and internal variable is involved in an invariant for a given output variable. Furthermore, the user can compare different sets of invariants, allowing comparison of different versions of a system in order to inspect if and how they differ from each other. The display shows invariants that appear in both sets, invariants that only appear in the first and invariants that only appear in the second set.

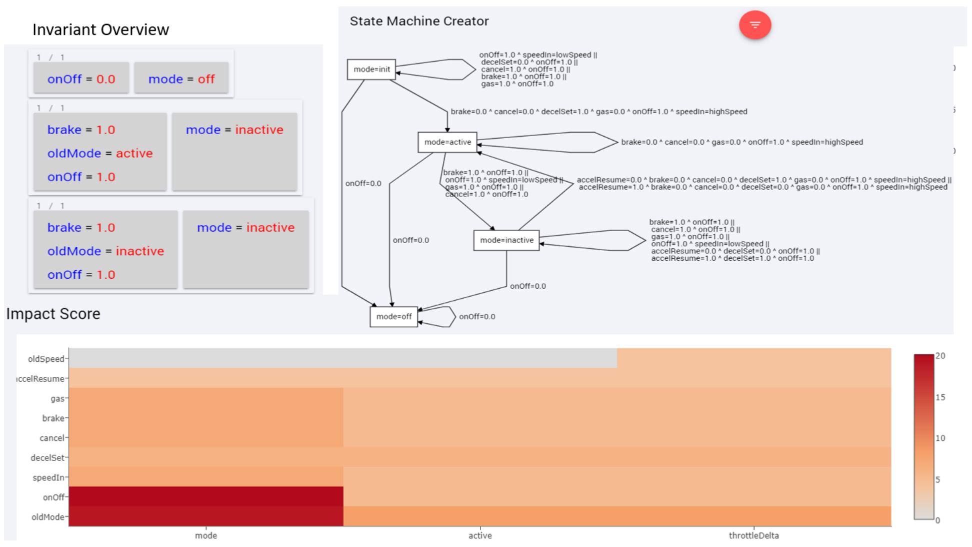

Invariant Overview
1 / 1
onOff = 0.0
mode = off
1 / 1
brake = 1.0
oldMode = active
onOff = 1.0
mode = inactive
1 / 1
brake = 1.0
oldMode = inactive
onOff = 1.0
mode = inactive
State Machine Creator
mode=init
onOff=1.0 ^ speedIn=lowSpeed || decelSet=0.0 ^ onOff=1.0 || cancel=1.0 ^ onOff=1.0 || brake=1.0 ^ onOff=1.0 || gas=1.0 ^ onOff=1.0
brake=0.0 ^ cancel=0.0 ^ decelSet=1.0 ^ gas=0.0 ^ onOff=1.0 ^ speedIn=highSpeed
mode=active
brake=0.0 ^ cancel=0.0 ^ gas=0.0 ^ onOff=1.0 ^ speedIn=highSpeed
brake=1.0 ^ onOff=1.0 || onOff=1.0 ^ speedIn=lowSpeed || gas=1.0 ^ onOff=1.0 || cancel=1.0 ^ onOff=1.0
accelResume=0.0 ^ brake=0.0 ^ cancel=0.0 ^ decelSet=1.0 ^ gas=0.0 ^ onOff=1.0 ^ speedIn=highSpeed || accelResume=1.0 ^ brake=0.0 ^ cancel=0.0 ^ decelSet=0.0 ^ gas=0.0 ^ onOff=1.0 ^ speedIn=highSpeed
onOff=0.0
onOff=0.0
mode=inactive
brake=1.0 ^ onOff=1.0 || cancel=1.0 ^ onOff=1.0 || gas=1.0 ^ onOff=1.0 || onOff=1.0 ^ speedIn=lowSpeed || accelResume=0.0 ^ decelSet=0.0 ^ onOff=1.0 || accelResume=1.0 ^ decelSet=1.0 ^ onOff=1.0
onOff=0.0
mode=off
onOff=0.0
Impact Score
oldSpeed
accelResume
gas
brake
cancel
decelSet
speedIn
onOff
oldMode
mode
active
throttleDelta
20
15
10
5
0

Figure 5.2: Example features of the Insight tool. The top left shows the invariant overview, the top right, the state machine creator and the bottom, the impact score heat map.

5.3 Shortcomings of the approach

A pilot study in [20] highlighted the utility of the invariant-generation approach discussed in the previous section: missing requirements in a specification of production automotive control software were uncovered. Other experiences with the tool similarly produced valuable insights into model behavior. However, in experiments we undertook on a variety of other models we encountered some limitations of the basic framework. Three of these issues — *Spurious Invariants*, *Explosion of the Search Space*, and *Continuous Variables* — became the motivation for this work. We describe these below.

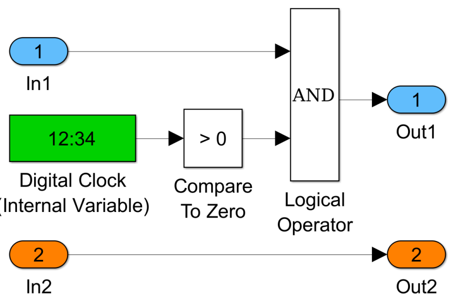


Figure 5.3: Example model illustrating a constellation that leads to spurious invariants

5.3.1 Spurious Invariants

While the retest cycle in [20] identifies many false positives, there are cases where the approach returns an objectively true invariant (one that cannot be disproven by the data) that gives misleading information about the behavior of the system (see next paragraph for an example). One of the reasons why this happens is that the association-rule mining process only identifies statistical relationships, not causal dependencies, between variables. As a side effect there are cases where the approach identifies invariants between variables that have no actual causal dependency in the system itself; we refer to these invariants as *spurious*. Figure 5.3 contains a simplified Simulink model that can give rise to spurious invariants. The system has two inputs, two outputs and an internal clock. The clock is used for the initialization step of the system and ensures that Out1 is zero after the system has booted up. Note that in our example input in2 is not connected to output out1; this is the intended functionality based on the requirements of the system. In these types of scenarios the approach infers spurious invariants relating values of in1 and out2, even though no connection exists between them. When spurious invariants are possible because of inputs that are unconnected to outputs in this fashion, they can overwhelm the actual invariants; in our experiments we observed ratios of actual invariants to spurious invariants as high as 1:100 depending on the number of in and outputs of the system. These spurious invariants make the analysis of the results much harder for the user and slow down the test generator considerably since it must process a larger number of invariants.

5.3.2 Explosion of the search space

This problem occurs in larger models that have many inputs, internal variables, and outputs. It leads to usage of large amounts of time and memory, and in our experiments even crashed due to lack of memory on the 32GB machine used during our evaluation.

To understand the source of the problem, consider the FP-Growth algorithm that we introduced in section 2.5. In order to create association rules it has to find all possible frequent item sets in the data, leading to a potentially very large search space which is exponential to the number of unique items in the data. The general complexity of this problem is NP-hard but for special cases (e.g. bounded minimum support) it has been shown to be in P [103]. To generate the item sets the algorithm creates several tree structures called *FP-Trees*; the number of trees is also proportional to the number of unique items in the test data. So the main driver of the performance (execution time and memory consumption) of the algorithm is the number of unique items that it has to consider. The frequent items in our data are based on the inputs, internal variables and outputs of the system and their associated values from the test data, which means that with each additional variable that we have to consider the unique items increases by the number of values that this variable can take.

Another factor increasing the number of item sets that we have to consider is the *support* setting. The pruning of items that do not have enough support can reduce the search space greatly. However, since we want to identify all invariants no matter how rare they are, we have to use a very low minimum support threshold. Our evaluation shows that there is a big trade-off between performance vs accuracy. One can greatly speed up the data-mining process and reduce memory usage, but this is at the expense of missing invariants. Furthermore, without a low support threshold there can be cases where the approach does not converge. It turns out that some invariants might fall below the support threshold in one iteration, only to later reappear again due to the way the test data changed between iterations. If the new test data in an iteration lifted the support value for a previously dropped invariant over the threshold again it would reappear. This fluctuation of invariants shows that, besides not finding all invariants, a support setting that is too high can lead to non-convergence of our approach, since the accuracy of the set of invariants may not be guaranteed to increase monotonically with the number of iterations.

5.3.3 Continuous Variables

The approach in the original form was not able to handle unconstrained continuous variables and only worked with categorical variables. We will explain this issue using the example of a cruise control system of a car, which is one of the systems used in the experimental evaluations. The cruise control system can control the speed of the car to keep it close to a user supplied value. It can only be active if the user activates the system while the car is driving faster than 25mph, which corresponds to the invariant $on = true \& speed > 25 \Rightarrow active = true$, if the speed is below or equal to 25mph the cruise control shall be deactivated ($speed <= 25 \Rightarrow active = false$). Regular association rule mining treats each numerical value of an integer or floating point variable as a separate category. E.g. it would create the invariants $speed{=}24.45 \Rightarrow mode{=}off$, $speed{=}24.451 \Rightarrow mode{=}off$ and so on. The validation step would then try to find counterexamples to these rules creating more test data and the subsequent data mining step would create even more invariants. This stops the approach from converging since it always can find more invariants in each step.

If the user has some preexisting knowledge about the system, for example if he is aware of the boundary of 25mph Specstractor allows the user to encode this information as data abstractions. Abstractions take the form of inequality formulas and labels that are automatically applied to the data if the corresponding formula evaluates to true. For example if the speed is smaller than 25mph the label low speed is applied, if it is higher the label high speed is applied. There are, however, two problems with this approach. Firstly the user already needs to be aware of some of the system's behavior or would need to be able to find this information in existing specifications. Secondly this type of partitioning of the data gets more and more complex if there are overlapping value ranges in the resulting invariants. For example if we assume that the boundary of 25mph were not fixed but could be changed by a user supplied input (userLimit) from 25mph to 35mph the invariants would look as following:

- $on = true \& userLimit = low \& speed > 25 \Rightarrow active = true$
- $on = true \& userLimit = high \& speed > 35 \Rightarrow active = true$
- $(speed <= 25 \& userLimit = low \Rightarrow active = false)$
- $(speed <= 35 \& userLimit = high \Rightarrow active = false)$

The user supplied abstraction would have to be aware of all these different partitions in the data and become more and more complex.

5.4 Addressing the Shortcomings

The issue of Spurious Invariants and the Explosion of the Search Space is addressed with the help of data- and control flow dependency information (Chapter 6). This information will allow us to eliminate spurious invariants and reduce the search space of the data mining process. For handling continuous variables we developed the Range Mining Algorithm that uses the iterative process of the framework to refine an initial automated discretization of the continuous variables (Chapter

7). The approach leverages the existing association rule mining algorithm and uses rule merging and test generation to identify accurate rules over continuous variables on a per variable basis. Which means that it does not identify a potential global partition that is similar to the user supplied abstraction and instead identifies per invariant partitions.

5.5 Summary

This chapter presented the Specstractor framework and tool chain for the automated extraction and analysis of system specifications in the form of system invariants. Specstractor employs an iterative approach for the extraction of invariants from automatically generated test cases using data-mining techniques. Furthermore, the chapter presented the Insight analysis tool, which allows the user to inspect sets of invariants and offers features for filtering and for creating high-level views of the resulting invariants, for example in the form of state machines. The chapter also presents current shortcomings of the approach that we will address in the following chapters.

Chapter 6: Better Invariants via Static Analysis

This chapter describes how the problems of *Spurious Invariants* and the *Explosion of the Search Space* that were introduced in chapter 5.3 may be addressed using static analysis [32]. The particular artifacts we focus on are Simulink®[1] models; such models are widely used in industries such as automotive as a basis for the generation of embedded code. We show that by adding simple data- and control flow analyses into our invariant-generation technique presented in chapter 5.1, we can dramatically improve both the efficiency of invariant mining and the accuracy of the invariants that are produced.

We examine two different uses of this information. The first approach introduces constraints in the FP-Growth algorithm as described in [104]. This leads to fewer recursive calls, thereby reducing the search space. Furthermore, all invariants that are produced this way only entail variables with actual causal dependencies, which will remove spurious invariants. The second approach partitions the test case data in such a way that for each output variable there is one dataset that only entails the inputs and internal variables that have a causal dependency to that output. The data-mining algorithm is then executed separately for each of these datasets. This

[1]Simulink® is a registered trademark of The MathWorks, Inc.

also reduces the search space and removes the spurious invariants.

The rest of this chapter describes our static analyses approach and how the information is factored into the data-mining process. We then develop an experimental framework for evaluating the new techniques using a corpus of 12 Simulink models of automotive and medical-device control systems, and demonstrate statistically how the static analyses lead to faster run times, lower memory usage, and better invariants. The final section discusses threats to the validity of our work.

6.1 Dependency Analysis

Our approach to pruning the search space of the invariant mining process and identifying spurious invariants involves determining which inputs and internal variables can affect the values of which outputs of a system. We compute this information by building a directed *dependency graph* from the Simulink model. Each input into the system, each read of an internal system variable, and each system output will be represented as a distinct vertex in this graph; computing causal information for a given output just involves performing a reachability calculation on this graph (e.g. using depth- or breadth-first search). The transformations discussed in this section are specific to Simulink models. However, the graph representation can be reused for other data-flow languages for which such dependencies can be computed. The rest of this section describes how this dependency graph is constructed for Simulink models.

Simulink models consist of *blocks*. Each block contains a collection of inputs and outputs together with logic describing how the block executes. Roughly speaking, the model of execution for a block is as follows. Inputs to the block are read when they are available, after which the block "fires" by executing its logic, thereby producing values on each of its outputs. Outputs of a block may in turn be connected to inputs of another block, inducing as a result a dependency between the blocks: the downstream block's outputs will depend in part on the outputs produced by the upstream block.

Simulink blocks may take different forms. So-called *basic blocks* are the smallest building blocks of Simulink models. Many such blocks, such as the Sum block, compute functions over their inputs, yielding results as outputs. Other blocks, such as Zero-Order Hold, have state that depends on previous values provided as inputs. Similarly, Data Store Read blocks permit internal variables created via Data Store Memory blocks to be read, while Data Store Write blocks allows these variables to be modified. (In this paper, for simplicity we assume that any value written to a Data Store is also output on some Outport. This simplifies our account of invariants as having "outputs" on the RHS.) Still other blocks are called *virtual*, as they are primarily responsible for routing data through a model. Examples of the latter include Inport and Outport blocks.

Other blocks take the form of *subsystems*, which contain submodels. The inputs and outputs of a subsystem correspond to the top-level Inport and Outport blocks of its submodel. In addition, certain subsystems also have special triggering inputs indicating when the model should, or should not, execute. In this regard, while Simulink is primarily a data-flow language, there are also control-flow aspects to its operational semantics.

In addition to basic blocks and subsystems, Simulink also includes blocks that may be used to include C code, or MATLAB code, or state machines in the Stateflow notation, within models. In this paper we will not consider these blocks.

6.1.0.1 Dependency Graph Construction

Our goal is to build a directed graph whose vertices are basic blocks as well as their outputs and inputs, and whose edges reflect data-flow / control-flow dependencies. We describe this construction in stages, beginning with generic basic-block assemblages and then continuing with our treatment of certain specialized blocks, including subsystems. Given space constraints, this discussion is necessarily abbreviated.

6.1.0.2 General Block Treatment

Dataflow between two basic Simulink blocks B_1 and B_2 is treated as in Figure 6.1. In this case, we introduce a vertices for B_1 and B_2 into our dependency graph, together with vertices for each output and input of the block. Outgoing edges in the figure connect B_1 to its outputs, while incoming edges connect B_2 to its inputs. Finally, any connections in the Simulink diagram between outputs of on block and inputs of another are added as directed edges in the obvious fashion.

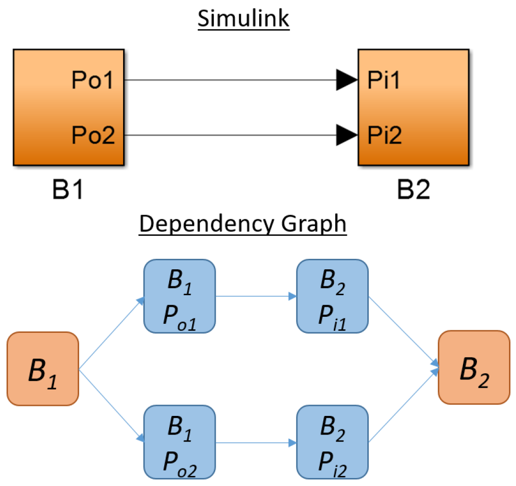


Figure 6.1: Transforming a Simulink block into a dependency graph

6.1.0.3 Virtual Blocks and Subsystems

Our dependency graph is intended to be a flattened representation showing causal connections between basic non-virtual blocks, as well as top-level Inports and Outports. The description below illustrates how we "translate away" subsystems and certain virtual blocks. We focus in particular on the following four cases: *From* and *GoTo* blocks; *Virtual Subsystems*; *Conditional Subsystems*; and *If Else* blocks. Graphical depictions of the corresponding transformations are given in Figure 6.2. Other types of virtual blocks and subsystems are treated similarly and are omitted.

GoTo/From. *From* and *GoTo* blocks (Example 1 in Figure 6.2) introduce virtual connections between blocks. The GoTo block GB_1 and the From blocks FB_1 and FB_2 in the example contain the tag *[A]*; this indicates that there are connections between the output of block B_1 and the inputs of B_2 and B_3. Our construction adds vertices for the GoTo/From blocks and directed edges from GoTo blocks to corresponding From blocks.

Subsystems. *Subsystem* blocks allow submodels to be embedded insider larger models while hiding the complexity of the former. Data is transferred into and out of the subsystem block via its submodel's Inports and Outports. Example 2 in Figure 6.2 depicts our treatment of Subsystem blocks. (The internal blocks of *Subsystem* SB_1 are overlaid in the figure for illustration purposes.) In the graph-construction process the Subsystem block in Simulink is omitted from the graph, but submodel is retained. Edges are introduced between the inputs and of the subsystem and their corresponding Inports and Outports from the submodel.

Conditional Subsystems. Some subsystem blocks (e.g. *Action Subsystem*, *Triggered Subsystem*) have a control-flow component to their behavior in addition to the dataflow component present in subsystems described above. Specifically, these subsystems contain *action / trigger ports* AP/TP; the subsystem is then only executed if the right trigger signal is supplied during run-time. (Trigger signals can be evaluations of if/else statements or rising/falling edges of signals, for example.) As an illustration of our treatment of these types of subsystems, consider the *If-Action Subsystem* in Example 3 in Figure 6.2. The If block supplies an action signal to the action port (here labeled `if{}`), which activates execution of the subsystem during the simulation step if the supplied value is "true". Our construction first treats a conditional subsystem like a regular subsystem with regard to inputs and outputs. It also creates a vertex for the action port, adds an incoming port edge to this vertex as indicated in the model, and adds an edge from the action-port vertex to *every* vertex corresponding to a block in the submodel. This is necessary since a subsystem could have e.g. Goto/From blocks that are affected by the trigger port but that do are not affected by the inputs and outputs of the subsystem.

If Blocks. *If* blocks can have multiple inputs and outputs. One output is labeled *if*; there are optionally several *else if* outputs and one *else* output. Each of these outputs has a corresponding logical expression and is connected to a conditional subsystem block. The If-block in the Example 4 in Figure 6.2 has two inports, u1 and u2, and one if, one else-if, and one else output signal. During execution the if-statement $u1 == 1$ is evaluated; if it is true the corresponding subsystem is executed. If it evaluates to false the else-if statement $u2 == 1$ is evaluated and its corresponding block is executed if it is true; otherwise the else statement's block is executed. In our graph construction, each logical statement is treated as a separate vertex (if, eif, e) that is connected to the corresponding subsystem. The logical expressions are then analyzed, starting at the if-statement. Each input that occurs in the logical expression of the if-statement is connected to the corresponding if-vertex. In the example the expression of the if-statement is $u1 == 1$; therefore an edge from the $u1$ port to the if-vertex is introduced. The else-if statement is $u2 == 1$. However, in order to reach this part of the execution, the if-statement has to be false. The statement could therefore be rewritten to $!(u1 == 1) \wedge (u2 == 1)$, showing that both u1 and u2 influence the statement; edges are thus introduced from the $u1$ and $u2$ inports to the eif-vertex. The else-statement does not introduce a new logical expression, but again in order to reach it both the if and else-if statements have to be false, which requires both inputs.

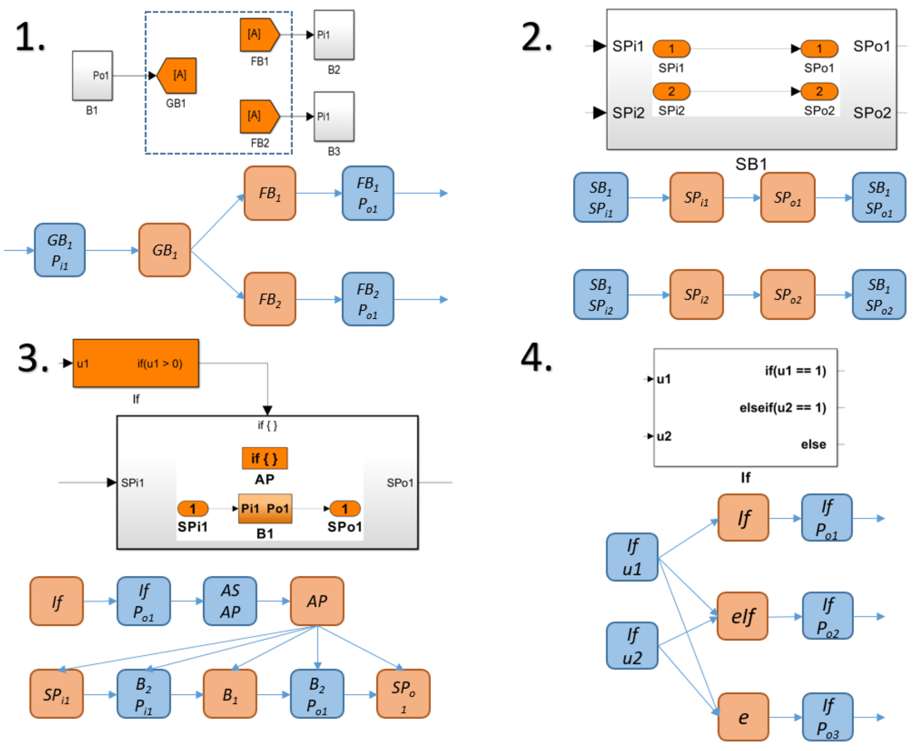


Figure 6.2: Graph constructions for 1) Goto/From, 2) Subsystem, 3) Conditional Subsystems and 4) If blocks

6.1.1 Association Rule Mining

We now explain our ideas for using the information stored in the dependency graph to improve the association-rule-mining component of the invariant-generation process. The basic idea is to ensure that for any association rule of form *LHS* $\Rightarrow$ *RHS* that is computed, there is a dependency between each proposition in LHS and RHS. If you recall section 2.5, the FP-Growth algorithm first identifies the frequent item sets in the test data and then generates the association rules from these item sets. It should be noted that a large proportion of the item sets that are generated cannot produce valid association rules for our purposes. For example, consider the model in Figure 5.3; based on our dependency information we know that in2 has no effect on out1, and therefore any association rule of form *in1* $\Rightarrow$ *out1* is spurious. Thus, an item set containing only {in2, out1} cannot be used to create a non-spurious invariant. The goal is then to make sure that these invalid item sets are not created in the first place, which also reduces the search space that the association-rule miner has to explore. We propose two different approaches to achieve this goal: *constraint-based mining* and *partitioned mining*. These are presented in more detail below.

6.1.1.1 Constraint-Based Mining

As a means to reduce the search space the FP-Growth algorithm permits users to define constraints on the item sets that are generated [104]. The constraints are generally either *monotone* or *anti-monotone* [105].

- A constraint is *monotone* if whenever set S violates the constraint, so do all subsets of S.
- A constraint is *anti-monotone* if whenever set S violates the constraint so do all supersets of S.

In what follows we show how to use the static-analysis information to create anti-monotone constraints that will remove all frequent item sets that result in association rules leading to spurious invariants. Since the FP-Growth algorithm creates item sets starting at size 1 up to the size of the longest transaction, the benefits of an anti-monotone constraint become clear. If one can rule out a set S one can immediately rule out any superset of S as a candidate set.

A valid association rule for our purposes has the form LHS $\Rightarrow$ RHS, where the LHS mentions inputs and internal variables and the RHS consists of a single output *that is dependent on every element in the LHS*. (We could allow more than one element in the RHS, but this reduces performance without enhancing the expressive power.) The following three constraints on item sets ensure adherence to a format that results only in valid association rules.

1. If an item set has more than one output variable (RHS item) in it, the set should not be saved or used to create larger item sets.
2. If an item set contains one RHS item and one LHS term that the RHS item is not dependent on, the set should not be saved or used to create larger item sets.
3. If an item set does not contain an output (RHS item) yet, and there exists no

output that is dependent on all the other (LHS) items in the set, the set should not be saved or used to create larger item sets.

It is easy to check that all three of these constraints are anti-monotone; any item set satisfying any of these constraints will also have all its supersets satisfying the same constraint.

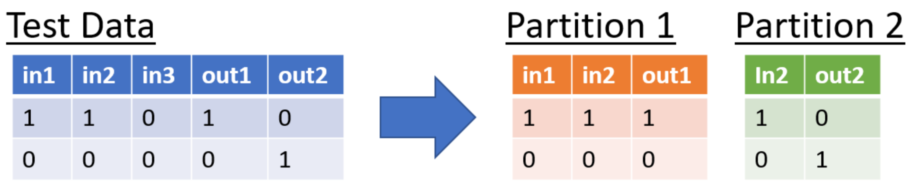
Rule 3 leads to the case where some item sets do not contain a RHS but are still saved. The reason for this is that it is necessary to have all subsets of a valid item set available in order to later calculate the confidence of the association rules. These incomplete rules are marked so that they are not considered for rule creation and will only be used for the confidence calculations.

Our constraint-based mining approach involves adding the three constraints just mentioned into the FP-Growth algorithm.

6.1.1.2 Partitioned Mining

In this approach, instead of modifying the mining algorithm as in the constraint-based mining approach, the input data is split into sets from which only non-spurious invariants can be generated. More specifically, for each output a subset of the data is extracted that contains the output and all inputs and internal variables that our static analysis indicates the output depends on. The FP-Growth algorithm is then applied separately on the subsets, and the resulting association rules of each partition are then aggregated.

Figure 6.3 shows how a sample test case might be partitioned. The test case


Test Data

in1	in2	in3	out1	out2
1	1	0	1	0
0	0	0	0	1

Partition 1

in1	in2	out1
1	1	1
0	0	0

Partition 2

In2	out2
1	0
0	1

Figure 6.3: Partitioning of the input data for the FP-Growth algorithm based on static analysis

has three inputs (in1, in2 and in3) (out1 and out2). Let us assume that static analysis has shown that in1 and in2 have has a connection to out1; in2 also has a connection to out2; and in3 is not connected to any of the outputs. The approach will therefore create a partition [in1, in2, out1] and a second partition [in2, out2]. To save memory the full test case data is stored once and the partitions only store the column indices of their associated inputs, internal variables, and outputs.

Partitioning the data beforehand leads to smaller data mining problems and, in principle, should therefore speed up the data mining process and reduce the memory consumption. This speed-up could potentially be offset by the fact that the algorithm must be invoked once for each output. The experimental section of the paper investigates this trade-off.

6.2 Experimental Evaluation

This section describes an experimental evaluation of our static-analysis-based approaches for improving invariant generation.

6.2.1 Hypotheses

The goal of the evaluation is to evaluate how well constraint- and partitioning-based data mining perform *vis à vis* the original approach outlined in Section 5.1, as well as with respect to each other. The following hypotheses guide our experiments.

- H1: Using dependency information produces higher accuracy invariants and fewer spurious invariants than not using it.
- H2: Using dependency information yields reduces the search space of the data mining process and improves performance in terms of execution time and memory consumption
- H3: The constraint-based approach and partition-based approach are indistinguishable in terms of accuracy and lack of spurious invariants, and also in terms of performance.

H1 is concerned with the accuracy of the resulting invariants and H2 with the performance of the static-analysis-based approaches in this paper in comparison with the original approach, which does not use this information. H3 is used to evaluate the two static-analysis-based approaches against each other.

Accuracy is mentioned in Hypotheses 1 and (indirectly) 3. Our intention here is to compare the automatically generated invariants against a collection of invariants, which we call the *golden invariants*, for each model in our study. These invariants were produced in a three-step approach. 1) the unimproved approach was applied to generate an initial set of invariants; 2) spurious invariants were then pruned using the static analysis information and 3) the resulting invariants were manually inspected to add missing invariants that should have been added and removing invariants that are invalid. The notion of *golden invariant* is somewhat subjective as a result, and the importance of the accuracy figures is thus relative (how well do the different techniques compare to each other?) rather than absolute.

For H2, performance is measured in terms of the total execution time of the overall process as well as the execution time of data mining, and in terms of the memory consumption during data ming. For the comparison of the improvements in H3 we hypothesize that the accuracy and lack of spurious invariants should be the same. For the performance we expect differences based on the characteristics of the model: we believe that models with fewer outputs should perform better in the partitioned approach, while models with many outputs should perform better in the constraint-based approach.

6.2.2 Variables and Data Collection

The dependent and independent (controlled and uncontrolled) variables for the experiments are listed in Table 6.1. The independent controlled variables include the test-generation settings of Reactis, which allow the user some control over how Reactis creates tests. For the purposes of this study we used two different settings. Generally speaking, Reactis combines random test generation followed by a targeted phase that supplements the randomly generated tests with new tests that improve coverage. In our first setting, we directed Reactis to create 100 random tests with

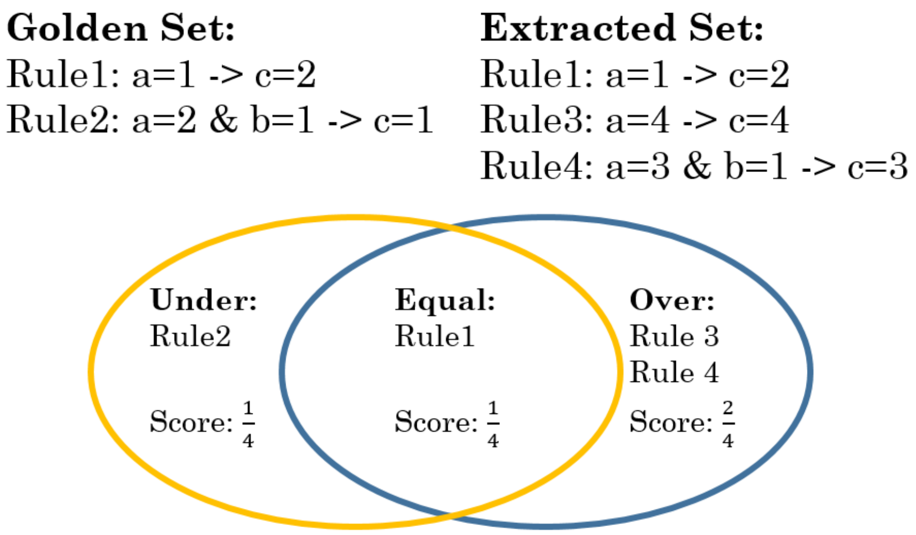


Figure 6.4: Illustration of the Jaccard- and the over/under score

100 steps each, and to use up to 20,000 execution steps in the targeted phase. The second test setting creates 1,000 random tests with 1,000 steps each, while again using 20,000 targeted steps. We will call the first test setting *short tests* and the second setting *long tests*. In both cases we selected all coverage criteria supported by Reactis; this means Reactis will attempt to achieve 100% coverage of all of these. Other controlled variables are the system models used in the evaluation and the settings of the requirement extraction process (e.g. turn the improvements on or off), as well as the minimum support threshold used in the data mining algorithms. The support threshold was set to 1 to guarantee that every invariant in the dataset is mined.

The dependent variables of the experiments are the runtime for each iteration in seconds (recorded for different parts of the process), the memory usage of the data-mining algorithm and the accuracy of the resulting invariants. For the execution time we record the overall time it takes to run the iteration as well as the time for the test generation, for the data mining and the time for the other processes (e.g. instrumentation). The time is captured by using a probes in the code. The peak memory usage of the data mining algorithms is measured using the Memory-Logger class of the SPMF data mining library. Spurious invariants are a subset of false positives and impact the accuracy of the resulting invariants. The accuracy is measured by comparing the resulting invariants to a set of semi-automatically created golden invariants of the systems using the Jaccard similarity coefficient [106]. The Jaccard coefficient is a set-similarity statistic that measures the overlap of two sets (the golden set of invariants, with the set produced by the framework), with a score of 0 (lowest) which signifies there is no overlap to a score of 1 (highest) which signifies complete overlap and therefore set equivalence. Two invariants are equal if both their LHS and RHS contain the same elements, any deviation (e.g. a=4 instead of a=1 on the lhs) means they are not equal and reduce the Jaccard score. However, a score below 1 can be caused by missing true invariants and/or false positives. We therefore added two more measurements based on the Jaccard coefficient. The under score indicates how much of the deviation from 1 is caused by missing invariants and the over score indicates how much is caused by false positives. This is illustrated in Figure 6.4; the golden set in this example contains two invariants, while the extracted set contains three. The intersection of the two sets contains only one invariant and the union of the set contains four invariants, and thus the Jaccard score of this example is 1/4. The under score is calculated by dividing the number of invariants in the complement *golden/extracted* with the elements in the union, which in the example is also 1/4. The over score is calculated accordingly using the complement $extracted/golden$ and yields $2/4 = 1/2$.

The independent uncontrolled variables are related to nondeterministic factors of the experimentation system, such as the load on the machine during the experiments, the available free memory, and the machine state. Another uncontrolled factor is the random aspect of the Reactis test generator, which is present in both the initial and targeted phases of the test-generation procedure. To control for these last aspects we constrain the random seeds used in the Reactis randomization process. Specifically, each experiment starts with the same fixed random seed and in each iteration the random seed is increased by one. This allows variations of test cases between the different iterations; however the random aspects of corresponding iterations in different experiments will be the same. The targeted test generation aspect depends on the instrumented invariants and cannot be fully controlled. To mitigate the impact of these uncontrolled variables, each experimental run is repeated ten times and the resulting dependent variables are then analyzed using their averaged values.

6.2.3 Test Applications

We use 12 Simulink models for our experiments. Ten of the models are from the automotive domain and represent parts of the lighting- and cruise-control systems of a car. The other two models are from the medical-device domain and are models of cardiac defibrillators [107]. The models vary in their complexity, with the smallest containing 40 Simulink blocks and the largest 627 blocks. The combined number of

Table 6.1: Un/Controlled and Dependent Variables of the Experiment

Type	Variables
Independent Controlled	Test Generator Settings, System Models, Approach (with/without improvements), Minimum Support Threshold
Independent Uncontrolled	Load on machine, Machine State, Free Memory, Random aspects of test generator
Dependent	Runtime in seconds, Memory Consumption, Jaccard Score

in/outputs and internal variables of the models varies from four to 57. From the 12 models five have internal state variables, while the other 7 are stateless. The number of golden invariants for the different models varies from 24 to 406. The last column shows the connectivity of the model, which is the maximum, minimum and average number of inputs and internal variables that the outputs of the system depend on. For example the Cruise Control model has two outputs; the first output is traceable via static analysis to all 9 inputs and internal variables, whereas the second one is only traceable to 8 of them, making its average number of connections from inputs to outputs 8.5.

6.3 Results and Discussion

The experimental data can be seen in Tables 6.3-6.6. The approach was executed with the three different data-mining strategies on each of the 12 models. For evaluation of the performance values we followed the example of Fontana [108] and calculated paired data by comparing the results of the unimproved approach (UA) against the constraint-based (CBA) and partitioning-based approach (PBA). The result is a percentage value that indicates the increase or decrease of time or memory consumption for each of the models.

The Jaccard score is of the resulting set of invariants after the iterative approach terminated. Because some experiments of the UA would not terminate properly, due to the spurious invariants, we introduced a 15-iteration limit. Execution times are given in seconds. Memory consumption is recorded in megabyte (MB).

Table 6.2: Model Metrics for the systems used in the evaluation. The column Variables lists the inputs, internal variables and outputs of the system.

Name	**Variables**	**Blocks**	**Invariants**	**Connectivity**
Cruise Control	7/2/2	83	34	9/8/8.5
Daytime Driving Light	14/0/3	52	31	9/8/8.3
Emergency Blinking	2/1/1	165	46	3/3/3.0
Exterior Light	31/4/22	515	406	17/6/11.5
Fog Light	10/3/4	59	70	10/7/8.8
High Beam Light	9/0/2	52	24	7/7/7.0
Low Beam Light	9/0/5	40	48	7/7/7.0
Parking Light	7/0/7	83	38	7/5/5.6
Position Light	9/0/7	48	30	5/5/5.0
Rear Fog Light	11/0/5	56	28	9/2/6.8
Defibrillator 1	4/3/3	184	63	7/7/7.0
Defibrillator 2	7/16/5	627	69	22/21/21.2

Table 6.3: Jaccard score and # of iterations to end the experiments.

	Short Tests: Jaccard (Iterations)			**Long Tests: Jaccard (Iterations)**		
Model Name	**UA**	**CBA**	**PBA**	**UA**	**CBA**	**PBA**
Cruise Control	0.72 ↑0.04 ↓0.34 15	0.93 ↑0.02 ↓0.05 12	0.92 ↑0.04 ↓0.05 13	0.94 ↑0.03 ↓0.03 (4)	**0.97 ↓0.03 (4)**	**0.97 ↓0.03 (4)**
Daytime Driving Light	T/O	1.00 (8)	1.00 (8)	T/O	**1.00 (4)**	**1.00 (4)**
Emergency Blinking	1.00 (4)	1.00 (4)	1.00 (4)	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**
Exterior Light	Crash	1.00 (7.3)	1.00 (7.1)	Crash	Crash	**1.00 (4)**
Fog Light	0.17 ↑0.83 (15)	0.97 ↑0.03 (5)	0.97 ↑0.03 (5)	1.00 (11)	**1.00 (4)**	**1.00 (4)**
High Beam Light	0.29 ↑0.71 (15)	1.00 (6)	1.00 (6)	1.00 (7)	**1.00 (4)**	**1.00 (4)**
Low Beam Light	0.62 ↑0.38 (12)	1.00 (6)	1.00 (6)	0.73 ↑0.27 (5)	**1.00 (4)**	**1.00 (4)**
Parking Light	1.00 (8)	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**
Position Light	0.42 ↑0.58 (12)	**1.00 (4)**	**1.00 (4)**	0.45 ↑0.55 (5)	**1.00 (4)**	**1.00 (4)**
Rear Fog Light	T/O	1.00/8	1.00 (8)	T/O	**1.00 (4)**	**1.00 (4)**
Defibrillator 1	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**	**1.00 (4)**
Defibrillator 2	Crash	Crash	**1.00 (5)**	Crash	Crash	Crash

Table 6.4: Data mining time per iteration in seconds

	Mining Time (Short Tests)					**Mining Time (Long Tests)**				
Model Name	**UA**	**CBA**		**PBA**		**UA**	**CBA**		**PBA**	
Cruise Control	**0.64**	0.78	122%	0.87	136%	**8.17**	8.62	106%	20.51	251%
Daytime Driving Light	T/O	1.19	N/A	**0.71**	N/A	T/O	**28.56**	N/A	47.15	N/A
Emergency Blinking	0.052	**0.035**	67%	0.068	131%	**2.49**	2.66	107%	4.37	176%
Exterior Light	Crash	560	N/A	**22.74**	N/A	Crash	21364	N/A	**560**	N/A
Fog Light	16.45	1.46	9%	**0.98**	6%	60.15	**17.12**	28%	34.35	57%
High Beam Light	0.84	**0.24**	29%	0.39	46%	18.84	**10.92**	58%	21.22	113%
Low Beam Light	0.33	0.2	61%	**0.87**	138%	15.9	**11.7**	74%	28.06	176%
Parking Light	0.33	**0.2**	61%	0.57	173%	12.45	**11.83**	95%	38.1	306%
Position Light	0.7	**0.18**	26%	0.45	64%	17.95	**14.59**	81%	32.75	182%
Rear Fog Light	T/O	9.56	N/A	**8.47**	N/A	Crash	**77.49**	N/A	153	N/A
Defibrillator 1	**0.12**	0.17	142%	0.45	375%	8.42	**8.41**	100%	26.56	315%
Defibrillator 2	Crash	Crash	N/A	296	N/A	Crash	Crash	N/A	Crash	N/A

Table 6.5: Average time per iteration in seconds

	Iteration Time (Short Tests)					**Iteration Time (Long Tests)**				
Model Name	**UA**	**CBA**		**PBA**		**UA**	**CBA**		**PBA**	
Cruise Control	95.48	**79.02**	83%	79.15	83%	**735**	737	100%	746	101%
Daytime Driving Light	T/O	61.40	N/A	**61.15**	N/A	T/O	**457**	N/A	458	N/A
Emergency Blinking	52.91	52.63	99%	**52.39**	99%	574	*573*	100%	575	100%
Exterior Light	Crash	874	N/A	**338**	N/A	Crash	26616	N/A	**4487**	N/A
Fog Light	316	66.70	21%	**66.67**	21%	1719	**813**	47%	836	49%
High Beam Light	138	58.18	42%	**57.95**	42%	824	**577**	70%	581	71%
Low Beam Light	102	**57.43**	56%	58.27	57%	709	**557**	81%	594	84
Parking Light	59.06	56.17	95%	**55.81**	94%	548	**546**	100%	571	104%
Position Light	64.11	49.57	77%	**49.37**	77%	438	**407**	93%	425	97%
Rear Fog Light	Crash	76.88	N/A	**76.76**	N/A	Crash	**824**	N/A	842	N/A
Defibrillator 1	68.86	**68.41**	99%	69.13	100%	**709**	710	100%	728	103%
Defibrillator 2	Crash	Crash	N/A	447	N/A	Crash	Crash	N/A	T/O	N/A

Table 6.6: Peak Memory Consumption during data mining

	Max Memory (Short Tests)					**Max Memory (Long Tests)**				
Model Name	**UA**	**CBA**		**PBA**		**UA**	**CBA**		**PBA**	
Cruise Control	550	520	95%	**324**	59%	7168	6421	90%	**5802**	81%
Daytime Driving Light	T/O	414	N/A	**200**	N/A	T/O	5759	N/A	**5543**	N/A
Emergency Blinking	**76**	335	441%	145	191%	**2189**	2468	113%	2313	106%
Exterior Light	Crash	5195	N/A	**2265**	N/A	Crash	Crash	N/A	**12862**	N/A
Fog Light	5051	316	6%	**194**	4%	11290	**6634**	59%	7009	62%
High Beam Light	416	177	43%	**132**	32%	6834	6480	95%	**6162**	90%
Low Beam Light	525	203	39%	**144**	27%	6422	6172	96%	**5950**	93%
Parking Light	260	163	63%	**106**	41%	Crash	**6170**	N/A	7694	N/A
Position Light	590	150	25%	**115**	19%	7127	**5568**	78%	5868	82%
Rear Fog Light	T/O	**256**	N/A	270	N/A	T/O	**6576**	N/A	6995	N/A
Defibrilator 1	241	267	111%	**228**	95%	4950	5932	120%	**4443**	90%
Defibrilator 2	Crash	T/O	N/A	**20953**	N/A	Crash	T/O	N/A	T/O	N/A

Experiments that crashed the experimental setup are marked with *crash*. A *T/O* marks an experiment that exceeded a limit (8 hours) that we set in place to cap the execution of an iteration. We omitted separate mention of the static-analysis times, since it has to be performed only once at the beginning of the process and the impact on the performance is minuscule compared to the testing and data mining. For example, the longest execution time of the dependency analysis was 73 milliseconds (for the Exterior Light model).

6.3.1 Hypothesis 1: Accuracy

For H1 we are evaluating whether or not the accuracy was improved with the introduction of CBA and PBA. Table 6.3 shows the Jaccard score after the last iteration of each experiment and how many iterations it took to finish. As can be seen, the UA crashed for two of the 12 models due to lack of memory (which means it needed more than the 32GB available to it) and timed out for two more. For the 10 models that it was able to process fully, in the short tests Reactis setting it has a lower accuracy in eight models and an equal accuracy in two. For the long tests the UA has a lower accuracy in six models and equal accuracy in four. This indicates that H1 is true, because the accuracy of the improvements is always higher or at least as high as the UA. The unimproved version often also requires more iterations than the improved versions, since the improvements can eliminate spurious invariants early on, meaning the test generator can focus on a smaller number of invariants.

The UA has issues with spurious invariants in five models for the short tests;

however for two of these models in the long tests the accuracy is the same as the improved version. This suggests that the issue with spurious invariants can be overcome, at least partly, with additional random test data, but it is not guaranteed. The results also seem to provide further evidence to the observation in [20] that better (in this case more tests) improve the accuracy.

6.3.2 Hypothesis 2: Performance

For H2 we are evaluating whether or not the performance in terms of execution time and maximum memory consumption was improved with the introduction of CBA and PBA. Table 6.4 shows the average execution times for the data-mining process in each iteration of an experiment; the average time for the full iteration (test generation + data mining + instrumentation) can be seen in Table 6.5 and the maximum memory consumption during data mining in Table 6.6. For the short tests the UA crashes for two out of 12 models; it times out for two more. It is slower in six and faster in two models. For the models where the UA does not crash CBA is 38% faster and PBA is 34% slower on average. For the long tests the UA also crashes and times out two times, respectively. It is slower in six and faster in two models. The CBA miner is 19% faster on average, and PBA 97% slower. The cases where the UA is competitive with CBA are mostly the small- to medium-sized models, with highly connected in/outputs. This is to be expected since the improvements are targeted at removing overhead from unconnected values. The slightly faster performance in some of the cases with high connectedness indicates that there is some overhead cost associated with the improvements. This was expected, especially for PBA, since it has to be executed once for every output. Also, checking the constraints for each item set in CBA incurs a cost that only pays off if it reduces the search space.

The overall time per iteration is the time spent on test generation, data mining and other tasks in the framework (e.g. instrumentation). For small- to medium-sized models with high connectivity the execution time of the data miner and the overall iteration time is similar (e.g. Parking Light). However, for models where the UA infers many spurious invariants (indicated by the low accuracy of the results, e.g. High Beam Light) the execution time of the iteration can be slowed down considerably. This is due to the fact that test generator has to spend time trying to disprove the spurious invariants. The *Daytime Driving Light* and *Rear Fog Light* are extreme cases for this behavior. They produce over 3000 spurious invariants, and the test generator therefore breaks the eight-hour limit that we set for each iteration.

The UA uses more memory in seven cases and less in one case for the short and long tests. On average CBA uses 3% more memory for the short and 7% less for the long tests. However, the CBA result is skewed by the one outlier in the *Emergency Blinking* model, which has a very low memory footprint. Without this model CBA is on average 46% faster for the short and 9% for the long test. PBA uses 59% less memory in the short test and 18% in the long tests. The crashes of the system are caused by lack of memory. These results show that H2 is not always true. For some cases where the system is highly connected there can be performance penalties due to the overhead associated with checking the constraints and a large overhead for the partitioning of the data that can make the improvements slightly slower. However, for the small- to medium-sized models the data-mining time is only a small factor of the overall time and even though there are improvements, they are negligible in comparison to the overall iteration time. But the improvements can have a significant impact for larger models both in terms of execution time and memory consumption. The unimproved approach was not able to handle the data mining for the two largest models. The two magnitudes of additional test data from the short to the long tests have a large impact on the execution time and the memory consumption of the approach. The FP-Growth algorithm only has to read the data twice and is therefore not bounded so much by the amount of data than other association-rule-mining algorithms. However, generating and parsing the tests can take considerable amounts of time for the larger tests. Also all the test data is aggregated in each iteration and is kept in memory which can also take up 2-7GB of memory (depending on the number of variables). One solution to this problem could be the addition of mining approaches that allow to add data incrementally; this way only the new test data would have to be read and stored in memory and could be discarded after the iteration.

6.3.3 Hypothesis 3: Improvement Comparison

For H3 we are evaluating whether or not CBA and PBA are performing the same in terms of accuracy and execution time/memory consumption. The results in Table 6.3 show that the accuracy of the two approaches is the same except for one case. In the short tests of the cruise-control model CBA has an accuracy of 0.95 and takes 12 iterations and PBA has an accuracy of 0.94 and takes 13 iterations. We analyzed the data and applied the test data from PBA to CBA but they produce the same results. However, what is different is the order in which the invariants are recorded. The order of the invariants could influence the generated tests from Reactis and explain this small discrepancy.

For the small- and medium-sized models (Table 6.4) CBA is usually better since PBA has to executed once for each output. This is especially apparent for the long tests. For the larger models, however, PBA performs much better. PBA is the only algorithm that can handle the second defibrillator model and for the Exterior Light model PBA mines the invariants in 23s for the short and 560s for the long tests. It takes CBA 560s and 21,070s respectively to mine the same data. Furthermore, PBA tends to use less memory than CBA, although this is not true for all models. Thus, the first part of H3 is true; CBA and PBA have the same accuracy. The performance of the two approaches depends heavily on the model characteristics and the amount of test data; also PBA tends do be less memory-intensive for systems that have less connectivity and therefore the second part of H3 is not true. Analysis of the connectedness could be used to determine dynamically which of the algorithms to use for a model. Except for fully connected models the static analysis information always proved useful. Adapting the idea behind CBA requires that the algorithm supports anti-monotone constraints and requires changes in their source code. The idea behind PBA on the other hand can be easily adapted to other data-mining algorithms. For future work we adapted it to a decision-tree-based algorithm, without having to make any changes to the algorithm itself.

6.4 Threats to Validity

The discussion about potential threats to validity of the experimental evaluation presented in this paper is based on Wohlin et al's [88] four-class layout that includes: threats to internal, external, construction, and conclusion validity.

6.4.1 Threats to Internal Validity

We considered two potential problems for internal validity. 1) Do the different implementations of the data miner (unimproved, constraint-based and partition-based) work as intended? 2) Are there any inefficiencies in the implementation that could explain the results? In order to evaluate if all the algorithms do the right thing we tested them with data for which we know the resulting invariants. To avoid accidental inefficiencies we used the same implementation of the FP-Growth algorithm for all three versions. The difference between them is a boolean flag that activates the additional constraint checks but otherwise they share the same code base. We also profiled the code with JProfiler [2] to identify potential bottle necks.

6.4.2 Threats to External Validity

The specific implementation of our approach in this paper analyzes Simulink models, and we applied it on models from the automotive and medical-device do-

[2]https://www.ej-technologies.com/products/jprofiler/overview.html

main. However, it should be applicable to other Simulink models as well. In fact, we believe that the approach is generally applicable as long as the requirements in Section 5.1 are met.

6.4.3 Threats to Construct Validity

Using incorrect measures in the experiments is a threat to construct validity. For the performance of the improvements we measured the time and the memory consumption, which are direct measures and therefore good indicators for relative performance. For measuring accuracy we use the Jaccard set similarity score to compare the generated set of invariants against a golden set we provided and validated as described in section 6.2.1. To make sure that the measures were not influenced by any uncontrolled factors (like the state of the experimentation machine) we repeated all experiments 10 times and averaged the results.

6.4.4 Threats to Conclusion Validity

In order to mitigate self bias, we applied the approach to a variety of models where some of the models played to the strength of the approaches (inputs and internal variables that are not connected to some of the outputs) and some represented the worst-case scenario (all inputs, internal variables are connected to all the outputs).

6.5 Conclusion

This chapter introduced an approach to identify data- and control flow dependencies and used the resulting dependency information to improve the Specstractor framework. The data- and control flow information was obtained by transforming a Simulink model into a directed graph using transformation rules presented in this work. The resulting graph could then be searched to identify the dependencies between the inputs, internal variables and outputs of a system. This information was used to introduce two improved versions of the data mining process of the specification extraction approach. The improvements reduce the false positives and the search space of the data miner, thereby increasing the accuracy and decreasing the execution time and memory consumption. The first improvement introduces constraints into the FP-Growth algorithm used in the approach during the item set generation phase. It removes item sets that would result in invalid association rules using the information from the static analysis. The second improvement partitions the data used in the FP-Growth algorithm so that it only contains outputs and associated inputs and internal variables that could be traced to each other via the dependency information.

We designed and executed an experimental evaluation of the improvements on 13 Simulink models from the automotive and medical device domain. The experiments show that the improvements made it possible to analyze 4 of the models that crashed without improvements. For the other models they could reduce the execution time and the memory consumption (by %26/%42). Furthermore, the improvements increased the accuracy of the approach by removing the false positives that were inferred by the data miner. The constraint based data miner was fastest for medium sized models and more test data, whereas the partition based data miner would have lower memory consumption and would scale better for larger models but has a larger overhead because it has to be executed once per output value.

Chapter 7: Mining Invariants from Continuous Variables

Traditional association rule-mining algorithms such as A Priori [101] and FP-Growth [35] do not work well with continuous and unbounded, as they treat each value of an integer or floating-point number separately. For example, for one of our models, a cruise-control system of a car, the approaches would create the invariants $speed = 24.4 \Rightarrow active = false$, $speed = 24.45 \Rightarrow active = false$ and so on. The validation step would then try to find counterexamples to each of these rules, creating more test data, and the subsequent data-mining step would find even more invariants. This stops the approach from converging since it can always find more invariants in each step.

If the user has pre-existing knowledge about the system, he/she can supply manually created abstractions for the data values that Specstractor would then automatically apply to the generated test data. For example if the user is aware of the boundary of 25 that determines if the cruise control can be active or not, he/she can create a mapping in the form of inequality formulas and labels that are automatically applied to the generated test data if the corresponding abstraction formula evaluates to true. For example, if the speed is smaller than 25 the label *low speed* might be applied, while if it is higher the label *high speed* is applied. This has the effect of collapsing the continuous speed variable into two abstract values.

There are, however, two problems with this approach. Firstly, the user already needs to be aware of some of the system's behavior or would need to be able to find this information in existing specifications or by inspecting the system manually. Secondly this type of partitioning of the data becomes more complex if there are overlapping value ranges in the resulting invariants. For example if we assume that the boundary of 25mph were not fixed but could be changed by a user supplied input (userLimit) from 25mph to 35mph the invariants would look as follows:

- $on = true \ \& \ userLimit = low \ \& \ speed > 25 \Rightarrow active = true$
- $on = true \ \& \ userLimit = high \ \& \ speed > 35 \Rightarrow active = true$
- $speed <= 25 \ \& \ userLimit = low \Rightarrow active = false$
- $speed <= 35 \ \& \ userLimit = high \Rightarrow active = false$

The user supplied abstraction would have to be aware of all these different partitions in the data. Quantitative association-rule-mining [83,84] algorithms have been proposed to infer association rules from continuous variables; however in our evaluations they did not perform well. We thus created an approach that builds on our existing iterative framework to infer accurate invariants over continuous variables.

7.1 Range Miner

Our Range Miner algorithm uses the existing iterative process of Figure 5.1 to refine an initial automated discretization of the continuous variables (see Figure 7.1). It starts by analyzing the first set of test cases that are generated and discretizing the value range of any continuous variable using modified frequency and equal length data binning. Invariants are then inferred from the binned data using the existing FP-Growth algorithm. Invariants with neighboring bins are merged [109] (see Figure 7.2) to reduce the number of invariants to verify.

Without manual abstractions each variable often can take many more different values (one for each bin), which leads to many more false positives. We therefore supplement the test generator to better cope with the additional false positives. This is achieved with additional test data generated by fuzzing test cases (see Figure 7.3) from earlier iterations and by carrying over information for the test generator between iterations. After the process terminates each invariant with a binned variable is analyzed to check if its boundaries can be extended without violating the invariant (see Figure 7.4). The remainder of the section explains the steps in more detail.

7.1.1 Discretization via Fixed Point Binning

Discretization via data binning is a standard method to pre-process data for data-mining algorithms [110]. Data values are put into bins, or buckets, that represent a range of values. Two common types of binning methods are *equal length* and *frequency binning* (see Figure 7.1). With frequency binning, values are put into bins

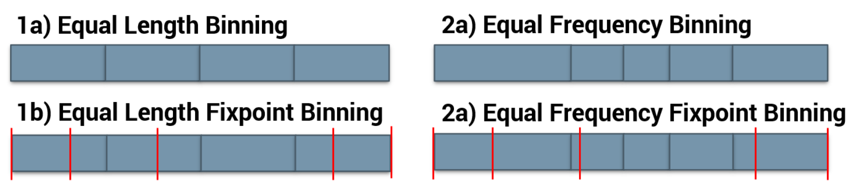

1a) Equal Length Binning
2a) Equal Frequency Binning
1b) Equal Length Fixpoint Binning
2a) Equal Frequency Fixpoint Binning

Figure 7.1: Discretization of the test data using equal frequency and equal length binning as well as fixpoint binning methods.

of equal number of elements, making for fine-grained bins in areas where the data is dense and for coarse-grained bins in areas where the data is sparse. Equal-length binning on the other hand divides the value range into bins of equal length and does not take density of the data into account.

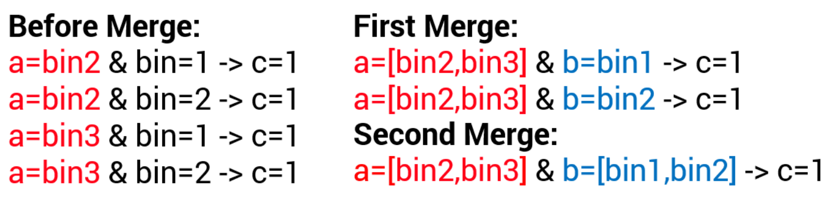
The equal-frequency binning method is making use of a characteristic of the generated test data of Reactis that makes it ideal for the binning of input values. Specifically, Reactis produces more input values that are close to interesting decision points in the system. For example, if there is a inequality check in the program (e.g. speed > 25) then there will be more values that are close to 25. The frequency-binning approach will thus automatically produce very fine-grained bins around 25 and coarse-grained bins in other parts of the system. However, it is still very unlikely that it will bin the data in a way that exactly hits the 25 threshold. This can be rectified by the final invariant adjustment of the Range Miner; moreover, by leveraging other characteristics of the Reactis test data, we can create a better initial binning that requires less adjustment.

We call this modified method Fixed-Point Frequency binning (see Figure 7.1). In addition to the bins created by frequency binning we add in bins that correspond to interesting points that we identify using heuristics on the generated test data. Reactis produces two kinds of interesting points. Firstly, frequency-based fixed points, the most interesting values are often the most frequent ones in the test data. Secondly, floating-point values generated by Reactis are rarely integers numbers except when they are interesting points, or very close to interesting points (+-1). Each interesting point will become a fixed bin, and will be added to the bins that were established by the binning algorithm. If the new bin is inside an existing bin the original bin is split. Even if the interesting points are not always accurate the approach will rectify the situation by merging them together with neighboring bins.

For outputs and internal variables frequency binning does not perform as well. This is due to the fact that the most frequent values that the internal variables and outputs take tend to coalesce around the most easily reachable parts of the system. Frequency binning would put interesting but rare output values into very large bins. For this reason we chose to use equal-length binning for internal variables and outputs. We modified the equal-length binner the same way we modified the frequency binner and added support for fix points.

7.1.2 Association Rule Mining and Merging of Invariants

The association-rule miner is unchanged apart from the fact that the test data is binned before it is handled by the miner. However, we add a merging step after invariant generation that performs an initial merging of the invariants. This step is very important for the performance of the test generator, which tends to slow down


Before Merge:
a=bin2 & bin=1 -> c=1
a=bin2 & bin=2 -> c=1
a=bin3 & bin=1 -> c=1
a=bin3 & bin=2 -> c=1

First Merge:
a=[bin2,bin3] & b=bin1 -> c=1
a=[bin2,bin3] & b=bin2 -> c=1

Second Merge:
a=[bin2,bin3] & b=[bin1,bin2] -> c=1

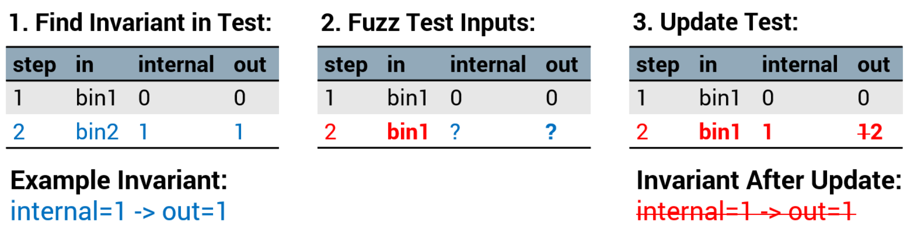
Figure 7.2: The datamining is performed on binned test data and the resulting invariants are merged as illustrated in this example.

if too many invariants are added. The merging approach is illustrated in Figure 7.2. It compares all the invariants of the same length and determines if they are equal in all parts except for one variable; if they are then they are merged together. For invariants that involve more than two variables, finding the optimal merging strategy becomes a multi-dimensional knapsack problem [111]. We use a greedy algorithm to try to find the optimal merge strategy in these cases.

7.1.3 Improving the Test Generator

Compared to using manual abstractions, the range miner has to deal with many more occurrences of false positives. In order to better identify and remove these additional false positives we modified the existing test-generation process by adding three mechanisms: preloading of existing test data; fuzz testing; and the generation barrier.

In each iteration of the Extractor the Reactis test generator always starts at 0% coverage, carrying over no information from previous iterations. This is very inefficient because we already have existing test data that can cover many parts of the system and allow Reactis to perform better in the targeted test-generation


1. Find Invariant in Test:

step	in	internal	out
1	bin1	0	0
2	bin2	1	1

Example Invariant:
internal=1 -> out=1

2. Fuzz Test Inputs:

step	in	internal	out
1	bin1	0	0
2	**bin1**	?	**?**

3. Update Test:

step	in	internal	out
1	bin1	0	0
2	**bin1**	**1**	**~~1~~2**

Invariant After Update:
~~internal=1 -> out=1~~

Figure 7.3: The datamining is performed on binned test data and the resulting invariants are merged as illustrated in this example.

process. Reactis has a feature that allows the preloading of an existing test data. However, if we would load all the existing test data in each iteration Reactis would have to process more and more test data in each iteration, which would bring it to a standstill after a few iterations. Instead of preloading all data we sample the existing test data. In particular, we search the test data for the first N occurrences of each identified invariant and then only extract the test data that is necessary to cover these invariants.

In order to invalidate an invariant the test generator has to find a state in a system execution in which the LHS of the invariant is true but the RHS is not. In deterministic systems this can only happen as follows. The system must be in a state where the LHS of the invariant is true and other inputs or internal variables that are not part of the LHS change the output of the system in a way so that it makes the RHS of the invariant false. We therefore have incorporated fuzz testing to help with the invalidation of false positives (see Figure 7.3). For each invariant the fuzz tester identifies N test steps that support the invariant. The test case is then copied from the start of the test case to the test step where the invariant is supported. The fuzzer then modifies input valued that are not part of the LHS of the invariant to identify a counter example to the invariant.

The generation barrier leverages an observation made during the evaluation of the range miner. We saw that the true invariants are almost always in the test data after two or three iterations. Any invariant appearing afterwards is nearly always a false positive. The generation barrier is a user-configurable limit that sets the number of iterations of the approach after which no new invariants are retained anymore. After the barrier has been reached additional iterations will only try to remove the remaining false positives. The addition of the generation barrier greatly improved the performance of Range Miner. Often, before the barrier is reached, the test generator would invalidate 30 invariants just for 30 new false positives to appear based on the new test data.

7.1.4 Final Adjustment

The buckets that are created by the automated binning step are not always accurately lined up with interesting decision points in the system. For example in one of our models the mined invariant *measuredValue = [130,180) & timer = [5,20]* $\Rightarrow$ *alarm = 1* states that if the measured value and the timer are in a certain range then the alarm is turned on. The invariant is correct but incomplete; the invariant describing the whole behavior of the system in this case would be *measuredValue = [130,180) & timer >= 5* $\Rightarrow$ *alarm = 1.* This invariant was not inferred since the maximum value the timer could reach during the text execution was 20 before

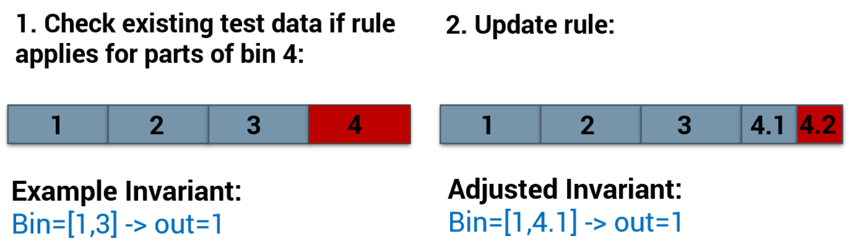


Figure 7.4: The data mining is performed on binned test data and the resulting invariants are merged as illustrated in this example.

the timer was either reset or the test case ended. In order to refine the resulting invariants we do a post-processing step after the invariant extraction has terminated.

In the post-processing step, we evaluate each invariant that contains a binned variable. We check if any of the binned variables can be extended into neighboring bins, or as was the case in the timer example, can be extended indefinitely, in case the bin is at the boundary of the observed values. To achieve this we create hypothesized invariants and test them against the accumulated test data generated during the invariant extraction. If the invariant cannot be extended into the whole neighboring bin, we create hypothesized invariants that sample the neighboring bin to determine if it is true for parts of the neighboring bin (See Figure 7.4). The invariants that could not be invalidated using existing data are again checked using the automated test generator.

7.2 Experimental Evaluation

This section describes the experimental evaluation of the Range Miner algorithm on 5 models containing continuous variables. The study was guided by 2 research questions.

- R1: How does the performance compare to manual abstractions?
- R2: Can it accurately extract invariants from continuous variables?

To answer the research questions we compare the execution time and resulting invariants of the our approach with invariants that we mined using manually-developed abstractions. This study is performed on 5 models using continuous variables. The experimental setup is the same as in the evaluation of the Static Analysis improvements in chapter 6.2, which describes in detail dependent and independent variables of the experimental approach and the data-collection mechanism.

7.2.1 Test Applications

Table 7.1 lists the 5 systems used in our evaluation: 4 model [20,32,112] the lighting- and cruise-control systems of a car, while one is a model of a blood-infusion pump [113]. The models vary in complexity from 52 to 375 Simulink Blocks and from 10 to 18 inputs, internal variables and outputs. The number of continuous variables varies from 1 to 5.

7.2.2 Results and Discussion

To answer research question R1 and R2 we evaluated the Range Miner (RM) algorithm and compared it with our existing manual abstraction (MA) approach. Table 7.2 shows the resulting invariants, the number of false positives, the average number of iterations and the average iteration time to perform the extraction (split

Table 7.1: Metrics of the systems used in the evaluation of the Range Miner algorithm. The columns Variables and Continuous contain the number input,internal and output variables

Name	**Variables**	**Continuous**	**Blocks**
Cruise Control	7/2/2	1/0/1	83
Emergency Blinking 2	5/3/2	0/2/1	375
Daytime Driving Light	14/1/3	0/1/0	52
Fog Light	10/3/4	0/1/0	59
Blood Pump	2/5/5	1/2/2	238

into test generation, data mining and other) for both RM and MA. RM requires more iterations and more test generation time per iteration and is therefore always slower than MA. This is due to the increased number of false positives that RM has to remove and the additional test data that we preload in the new test generation process. The test-execution time tends to be the dominating factor in the Spec-stractor approach and it is influenced mostly by the number of invariants to verify. In the future we are planing to parallelize the test execution and perform multiple simultaneous executions over which the invariants are divided. This should speed up the test generation process greatly.

The number of invariants differs between MA and RM, but the difference comes from differences in the way that MA and RM partition the data. By merging

Table 7.2: Comparison between the manual abstraction and Range Miner algorithm

Name	**Invariants**		**False Positives**		**#Iterations**		**Iteration Time**		**Test Time**		**Data Mining Time**		**Time Other**	
	MA	**RM**	**MA**	**RM**	**MA**	**RM**	**MA**	**RM**	**MA**	**RM**	**MA**	**RM**	**MA**	**RM**
Cruise Control	31	29	0	2	5	11	92	188	84	131	6	54	2	3
Emergency Blinking 2	58	49	0	1	6	8	111	121	103	110	7	10	1	1
Daytime Driving Light	31	31	0	0	5	11	89	139	70	72	16	63	3	4
Fog Light	72	70	0	0	5	9	114	170	109	160	2	3	3	7
Blood Pump	39	30	0	1	5.2	12.2	122	166	109	124	11	39	2	3

values together the RM merged together behaviors that required two invariants to describe with MA. We did observed four false positives in RM that the test generator was unable to remove consistently (two in Cruise Control and one in Emergency Blinking and Blood Pump). During the ten repetitions of the experiments they appeared 6-7 times. This shows that the test generator is able to remove it but cannot do so consistently. Increasing the number of test cases (1000 instead of the default 100) and more targeted steps (50000 instead of 20000) in the test-generation process makes it more likely to remove all false positives, but did not guarantee it. However, with these test settings the execution time increased by a factor of six per iteration, and more iterations were required on average.

The results show that we can accurately infer invariants over continuous variables, but in order to do so we need more compute time. However, it also shows that there are still a few false positives that the test generator could not remove. In future work we are planing to address this issue by using an additional verification step with a model-checking tool such as Simulink DesignVerifier in order to check if it can identify the remaining false positive.

7.3 Conclusion

This chapter presented and evaluated the Range Miner algorithm for the inference of accurate invariants over continuous variables. The Range Miner algorithm leverages the existing iterative process of Specstractor together with automated discretization and consecutive rule merging to learn invariants in the form of $\mathit{speed} < \mathit{25} \Rightarrow \mathit{cruiseControl} = \mathit{off}$. In order to counter the additional false positives that appear in this approach we improved the test generation process. We preload and mutate a subset of the old test data so that Reactis can better target the false positives.

The evaluation of the Range Miner has shown that it can accurately infer invariants from variables with continuous values. However, it requires more test generation time and more iterations to invalidate additional false positives that do not appear when using manual abstractions.

Chapter 8: Identifying and documenting ONBs with Specstractor

In this chapter we present a case study that evaluates the Specstractor toolchain by applying it to Simulink models of automotive and medical devices. The extracted invariants are compared against the existing specification artifacts in order to identify ONBs in these systems.

The study shows how the resulting invariants of the Specstractor approach may be used to identify ONBs. We show how issues in the requirements documentation and deviations between the requirements and the system manifest in the invariants. We further show in which cases the approach is unable to identify requirements accurately and why it is unable to do so.

Lastly, we present effort data to demonstrate how much time it takes a tester to apply the approach to compare the resulting invariants against real world specifications artifacts.

8.1 Research Questions

The experimental evaluation of Specstractor was guided by the following 4 research questions.

- R1: How do the extracted invariants relate to requirements of a system?

- R2: How do ONB manifest in the invariants?
- R3: When are we unable to identify requirements of a system?
- R4: Using the approach, what is the effort to debug requirements?

To answer the research questions we applied Specstractor to 11 models of automotive-control systems and medical devices and compared the resulting invariants with the natural language requirements and state machine specifications of these models. The experimental setup is the same as in the evaluation of the Static Analysis improvements in chapter 6.2, which describes in detail dependent and independent variables of the experimental approach and the data-collection mechanism.

8.2 Test Applications

Table 8.1 lists the 11 systems used in our evaluation: 10 model [20,32,112] the lighting systems of a car, while one is a model of a blood-infusion pump [113]. The *Variables* column contains the number of inputs, internal variables and outputs. The *Continuous* column indicates which of the inputs, internal variables and outputs are continuously valued. The *Blocks* column shows how many Simulink blocks are in the models. The remaining two columns show what kind of specification the models have and how complex the specifications are. Eight of the models have natural-language requirements descriptions ranging from two to five pages, for an overall total of 30 pages. Manual analysis of these yielded a total of 70 requirements (3-17 per model). Three of the models have specifications in the form of state machines.

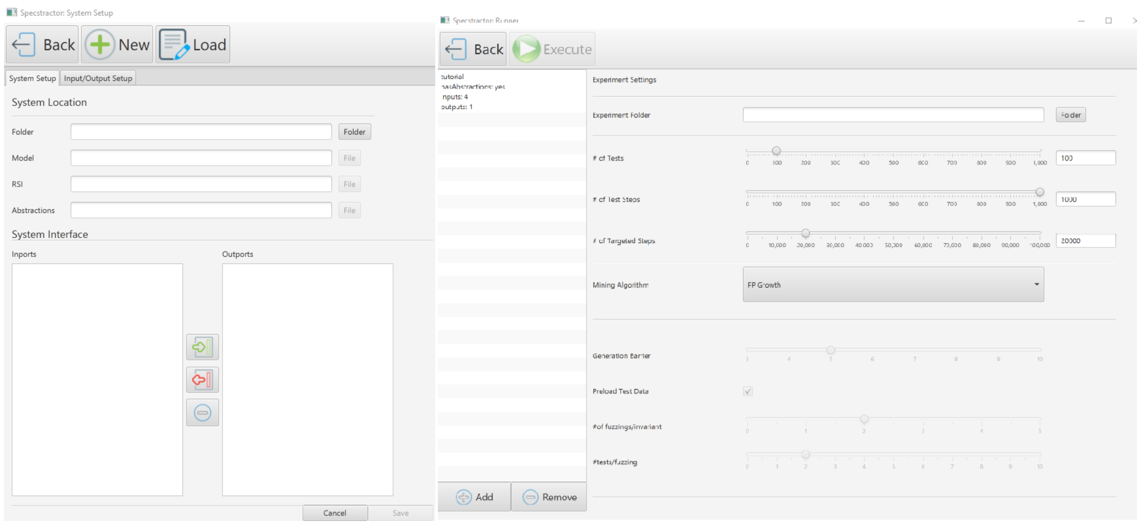
Table 8.1: Metrics of the systems used in the Specstractor evaluation. The columns Variables and Continuous contain the number input,internal and output variables.

Name	**Variables**	**Continuous**	**Blocks**	**Requirements**	**Requirement Complexity**
Emergency Blinking 1	2/1/1	N/A	165	State Machine	5 states/14 transitions
Emergency Blinking 2	5/3/2	0/2/1	375	State Machine	6 states/16 transitions
Emergency Blinking 3	5/1/3	N/A	107	State Machine	4 states/9 transitions
Daytime Driving Light	14/1/3	0/1/0	52	Natural Language	5 pages/7 requirements
Fog Light	10/3/4	0/1/0	59	Natural Language	3.5 pages/10 requirements
High Beam Light	9/0/2	N/A	52	Natural Language	3 pages/3 requirements
Low Beam Light	9/0/5	N/A	40	Natural Language	4 pages/6 requirements
Parking Light	7/0/7	N/A	83	Natural Language	4 pages/9 Requirements
Position Light	9/0/7	N/A	48	Natural Language	3 pages/6 Requirements
Rear Fog Light	11/0/5	N/A	56	Natural Language	4.5 pages/12 Requirements
Blood Pump	2/5/5	1/2/2	238	Natural Language	3 pages/17 Requirements

We counted each transition and state in a state machine as one requirement (4-6 states per model, with 9-16 transitions), for a total 54 requirements.

8.3 Specstractor work-flow

The work-flow of applying Specstractor to debug existing Specifications is as follows. The user starts by preparing the models for use in the Specstractor tool. In case that the system contains internal states it has to be instrumented so that these external states are accessible to Reactis. This step currently has to be performed manually and it takes about 10-15 minutes for a trainer user. We are planing to automate this step in the future.



tutorial
hasAbstractions: yes
inputs: 4
outputs: 1

# of Tests: 100
# of Test Steps: 1000
# of Targeted Steps: 20000
Mining Algorithm: FP Growth

Figure 8.1: Screenshots of the system setup and execution screen from the Extractor tool.

The user then opens the Extractor tool and starts the system setup. In the system setup (see left screenshot in figure 8.1) the user has to identify the system folder, the system files, the corresponding Reactis file and then choose the system variables that should be observed. Optionally if the user wishes to use manual abstractions an abstraction file can be provided at this point. After the setup is finished the user changes to the execution screen (see right screenshot in figure 8.1). On this screen the user can choose systems to execute, set the Specstractor settings (test generation settings, data mining algorithm), and then start the extraction process.

In order to compare the resulting invariants against the existing specifications the tester used the Insight tool and opened it side by side with a copy of the requirements documentation or state machine model. The tester would then analyze the document. Whenever he would see an requirement that describes the behavior of the system he would identify the involved variables and search for them in the Insight tool. This way he would see all invariants that are potentially related to a requirement. The tester would then take a screen shot of the invariants and copy it into the documentation. In case that the invariants are in agreement with the requirements he would mark the requirement green. If there were any deviations the user marked the requirement in red and commented on the deviation. Any parts of the requirements documentation that could not be mapped to invariants was marked purple which means there are missing invariants that could not be inferred or the requirement is not implemented. All unmapped invariants left from this process were then potential missing specifications or false positives and would undergo further evaluation by comparing them against the system implementation.

8.4 Results and Discussion

To answer R1-R3 we evaluated how the extracted invariants relate to the specifications of a system. To achieve this we applied the approach as described in the previous section and compared the resulting invariants manually against the system specifications. To answer question R4 we also recorded the effort that is necessary to map the resulting invariants to the specifications. For the systems with continuous variables we applied the Range Miner (see chapter 7.1) algorithm to mine invariants.

Table 8.2 contains the data from the evaluation for each model that was used including system name, number of unmerged and merged invariants. For perfor-

Table 8.2: Result of the mapping analysis.

Name	Invariants	Merged	False Positive	Req.	Acc.	Inacc.	Undoc.	Not Extracted	Expected To Miss	Effort
Emergency Blinking 1	46	46	0	19	19	0	7	0	0	90
Emergency Blinking 2	49	29	1	22	18	4	0	0	0	60
Emergency Blinking 3	65	43	0	13	13	0	0	0	0	60
Daytime Driving Light	31	30	0	7	3	3	0	0	1	40
Fog Light	72	39	0	12	6	3	0	3	0	120
High Beam Light	42	21	0	6	1	5	0	0	0	45
Low Beam Light	48	12	0	6	0	5	0	0	1	40
Parking Light	38	9	0	6	3	3	0	0	0	40
Position Light	30	5	0	5	1	3	0	0	1	30
Rear Fog Light	66	28	0	14	6	6	0	0	2	90
Blood Pump	39	28	1	16	6	7	0	0	3	80
SUM	526	290	2	126	76	39	7	3	8	695

mance reasons we configured the data miner in Specstractor so that each RHS can only have on value, e.g. $in{=}1 \Rightarrow out1 = 1$, or $in = 1 \Rightarrow out2 = 1$. To compare the invariants better against the requirements we merged rules with the same LHS values after the extraction has finished. The two above-mentioned rules would be turned into the following single rule: $in = 1 \Rightarrow out1 = 1$ & $out2 = 1$. The table includes the number of false-positive invariants and number of associated requirements of the system and the results of mapping the invariants to the requirements and the effort it took to perform the mapping (in minutes). To map the invariants to requirements we created the following classification schema.

Accurate Specification. These are specifications that can be completely described by the extracted invariants. An example is the invariant $brake = 1 \Rightarrow$

active = 0 that exactly matches the requirement *"The brake shall always deactivate the cruise control"*. Based on our definition of ONBs in chapter 1.1 all behaviors covered by these requirements and invariants are *Good Behaviors*.

Inaccurate Specification. Such specifications are not accurately reflected in the extracted invariants. An example can be seen on the left of Figure 8.2. The specification in this example states that *"If Blood Pressure (CNAP) improves within 10 seconds of the silent warning alarm, then the system will clear the alarm, and resume blood infusion."*. However, one of the corresponding invariants states (first on left) that whenever the blood pressure improves the warning alarm is cleared, independent of the time. The last part of this requirement contains another inaccuracy, as it states that in addition to clearing the alarm it shall resume blood infusion. As can be seen in the bottom two invariants the CNAP value is not the only value that decides whether the pump can be active or not. It also requires that either the reset button is pressed or that the critical stop flag has been cleared. The critical stop flag is set in case the patient is in a critical state and prevents automatic reactivation of the blood infusion even after the alarm cleared. If this happens the user has to press the reset button manually to restart the infusion, which contradicts the earlier requirement. This class of finding corresponds to either the *Bug/Defect* category or the *Good Surprise* category depending on the impact on the system. If the impact to the system is detrimental these would be counted towards *Bug/Defects*. If they describe valid but undocumented or partially wrongly documented behaviors they would count towards *Good Surprises*. We do not have contact to the original developers of the systems anymore, so we could not make this judgment ourselves.

Undocumented Behavior. This is system behavior that is not documented and does not contradict other existing specifications but that is described by the extracted invariants. An example is illustrated in the middle of Figure 8.2. The example is based on the *Emergency Blinking 1* system. With the help of the extracted invariants we were able to identify additional transitions (red) that were implemented but not specified. Again since we did not have access to the original developers of the systems, we could not judge if the undocumented behavior should be counted towards the *Good Surprise* or *Bad Surprise* category.

Not Extracted. These are behaviors of the system that is not covered by any invariant. An example can be seen on the right in Figure 8.2. Parts of the outputs of one of the systems where connected to a constant value that is not governed by any input or internal variable. However, our mining algorithm can only infer invariants that have a LHS and RHS and can therefore not infer an invariant like out=0. For the Fog Light system the approach was unable to extract an invariant for a specification that was caused by a missing instrumentation of an internal variable. The system used a Simulink block unfamiliar to the tester that internally used a memory block. Since the approach can only infer behaviors that it can observe it was unable to extract an invariant in this case. After fixing the instrumentation the invariant could be extracted. This issue motivates us to automate the instrumentation of internal system states in the future.

Partially Extracted. Such specifications are partially extracted. This is the case if only pieces of a specification show up as invariants and the rest are missing, or if an invariant has a small inaccuracy (e.g example $speed < 24 \Rightarrow active = 0$ instead of *speed* < 25). We never encountered this class in the resulting invariants in our study. Any such inaccuracy that appeared in continuous variables were corrected by the post-processing step described in chapter 7.1.4).

False Positive. This is an invariant that the approach extracted and which is not a true invariant for the system. Using manual abstractions we have not encountered false positives for the evaluation systems. However, a few false positives remained while using the Range Miner algorithm (see chapter 7.2.2).

Expected to miss/out of scope. Any specification that involves time or an internal state of the system where the time/state cannot be observed would fall in this category, as would any unimplemented features mentioned in the specifications.

In the three systems that were specified as state machines, 50 out of 54 states and transitions were accurate. We identified four violations and 12 invariants that described accurate but undocumented behaviors. A detailed analysis of the undocumented behaviors (see Figure 8.3) showed that the invariants correspond to one state and six transitions that were implemented but not specified. Two of the extra transitions can be reached and triggered using only nominal input values of the system (see left graph in Figure 8.3), while the other four transitions and the extra state can only be reached and triggered by using off-nominal input values that are not mentioned in the original specification of the system (seeright graph in Figure 8.3. The extra transitions and state seem to be implementing an undocumented error handling mechanism that deactivates the system. The off-nominal input leads to the error state and from there to the system off state. The ONB seems to be implementing valid behaviors and we would therefore classify them as a *Good Sur-*

prise. This shows that by letting the test generator choose off-nominal input values we can detect and document additional ONBs. None of the other classes of our schema were applicable for the state machine based models.

For the eight systems that are based on natural language requirements we counted a total of 72 requirements. Out of these 72, 26 were accurate and 35 inaccurate. Eight were out of scope and the approach missed to infer four (see classification for example). One was missed because the data mining algorithm cannot infer a static rule without a LHS and the other three because of a faulty manual instrumentation of the system. We intend to address both of these issues in future work. For static rules we plan to check if there are no changes in the test data for a specific value. Furthermore, we are planing to automate the instrumentation process to identify internal variables of the system. We expected to miss two of the requirements. The Daytime Driving Light system contains has a timing requirement (300 ms after action a, action b has to happen), and the Blood Pump contains a requirement that requires a system shutdown in case that values are out of range. Since the model component for system shutdown is not supported by the Reactis we removed this part.

The effort to analyze the resulting invariants varied between 30 and 120 minutes per model and on average was 63 minutes. This time includes the comparison of the invariants against the requirements as well as a manual inspection of the system to determine whether an invariant describes an undocumented behavior or is a false positive. It does not include the computer time necessary to extract the invariants nor the time to instrument the internal variables (10-15 minutes per model). For

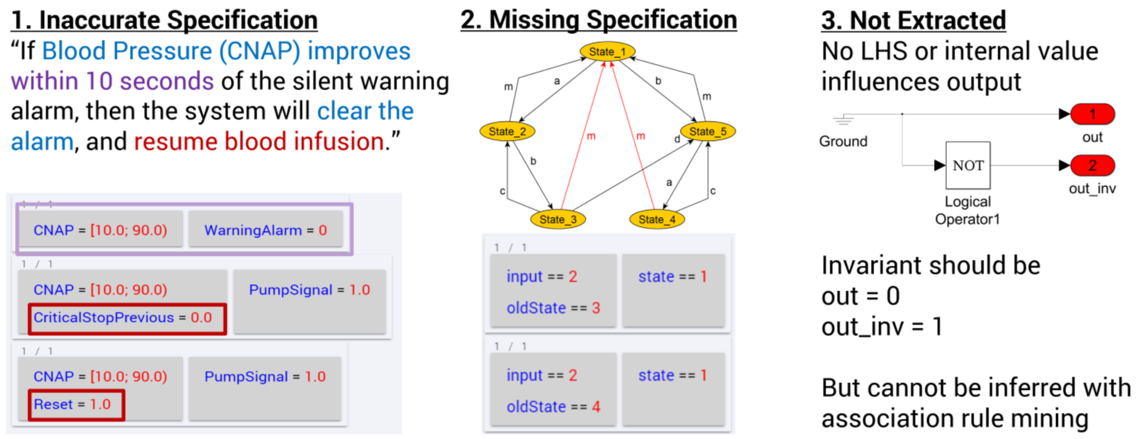


Figure 8.2: Three examples classes of findings from our case study for R1

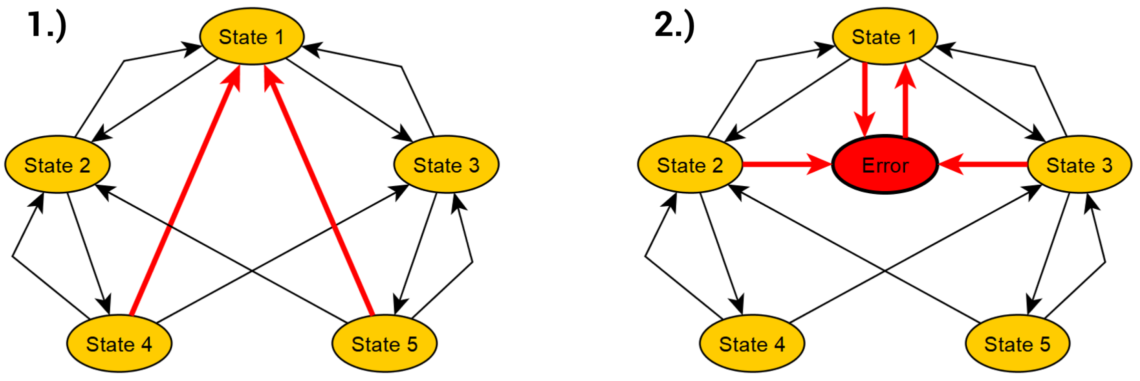


Figure 8.3: Examples of missing behaviors. The example on the left shows missing behaviors that could be identified with nominal input values. The example on the right shows missing behaviors that could only be identified with off-nominal inputs.

more information about the execution time of the approach we refer the reader our previous case study in [32], which contains a detailed execution time and memory consumption analysis. The most important factor in the effort to apply the approach was the number and complexity of the resulting invariants and of the requirements. One of the hardest invariants to verify and map manually was in the *Emergency Blinking 2* system. The invariant describes an emergent behavior in the state machine that for a given specific input the output is always the same independent of the current state. It required us to compare this specific invariant to all states and transitions. In all other cases the invariants contained an internal state variable with which we could restrict the verification to one state in the specification and its outgoing transitions. The most effort spent per model was on the *Fog Light* system. This was due to the fact that the tester missed instrumenting an unfamiliar Simulink block. Analyzing why this behavior was missed took about 50% of the analysis effort. Similarly, analyzing the undocumented behavior in Emergency Blinking 1 was very time-consuming in comparison to checking the invariants against the specifications. In its current form one of the downsides of the mapping approach is that any changes in the system can change the invariants and require a re-investigation. An important feature to make this approach more robust in the future would be to introduce actual tracing information into the requirements documentation, similar to features in Simulink and Reactis, that allow two-way links between model elements and existing requirements artifacts. This way when invariants are regenerated and they correspond to already linked invariants the links would stay; if invariants disappear one could identify broken links.

8.5 Conclusion

This chapter presented a case study of the Specstractor framework and its associated tool chain. In the case study we applied Specstractor to 11 systems of the automotive and medical device domain. In order to identify ONBs the extracted invariants were compared against the specification artifacts of these systems. The specification artifacts were in the form of natural language requirements (8 systems) and state machine specifications (3 systems).

We demonstrated that the approach can accurately infer invariants that describe the system with few false positives and these invariants can be used to identify ONBs that manifested in the form of mismatches between the specification and the implementation and missing specifications. The study showed that the resulting invariants gave very valuable insight into the system behavior and was able to detect many ONBs, in all three categories of ONBs. Specstractor seems to perform better for finding Good and Bad Surprises than MBT. This is most likely due to the fact that the Reactis test generator tries to cover the behaviors of the implementation of the system based on structural coverage criteria that are independent of the specifications of the system. Lastly, we present effort data which demonstrated that the comparison of the invariants against the specifications could be done in a reasonable amount of time.

Chapter 9: Conclusion and Future Work

This dissertation presented two areas of research with the goal to identify and document ONBs using artifacts available during the testing phase (e.g. models, source code, and executable of the system, as well as execution data from test executions and log files).

The first area of research was focused on the evaluation and improvement of state-machine based MBT for identification of ONBs. This research is guided by the first two of our three research hypotheses. To address our first research hypothesis *"MBT can be used to identify and test ONBs in software systems"* we evaluated MBT in a series of case studies to determine its effectiveness and efficiency. We showed that MBT could find several additional ONBs in already well tested systems but it required skilled testers and effort to create the testing models and their associated test infrastructure. MBT was especially good at finding ONBs in the Bug/Defect category which is not unexpected, since the models are based on the requirements. In order to identify Good/Bad surprises the tester often has to manually add behaviors and input data to the testing model, based on their experience and knowledge about the domain.

The second research hypothesis states *"Manually created test artifacts can be* *used to automatically infer testing models and their associated test infrastructure. The test cases generated from these models can identify ONBs that the original tests missed"*. The idea to leverage existing test cases is based on the lessons learned during our case studies. Existing manual test cases of the systems were very valuable for the modeling and test infrastructure creation process, both of which are very time-consuming aspects of the MBT process and also require skilled testers. We developed and evaluated an approach that leverages manually created test cases to improve the MBT process. The approach infers initial models and their associated test infrastructure from manual tests using heuristic transformation rules and the structure of the SUT. By automatically creating initial models we are addressing both the effort to create the model itself an its test infrastructure. We applied the approach in a case study and showed how the resulting tests from the generated models identified additional ONBs in the system that the manual tests missed.

The second area of research was guided by our third research hypothesis: *"Through Test Generation and Data Mining Techniques characterizations for ONBs can be identified and used to improve the requirements."*. To address this hypothesis we developed and evaluated Specstractor, a framework for the extraction of specifications in the form of system invariants from automatically generated test cases using data-mining. These invariants can convey information about system behavior and help us identify and document ONBs. We integrated static analysis information in the form of data- and control-flow analysis in order to remove false positives and increase the performance of Specstractor. Furthermore, we developed an algorithm that can mine association-rule invariants over continuously valued variables. We then applied the approach to several systems of the automotive and medical-device domain and compared the resulting invariants against the requirements artifacts of the systems. In this evaluation we showed how different types of ONBs can be identified and documented with the help of the invariants and that the resulting invariants have few false positives. Specstractor was good at finding all three types of ONBs. The reason for this lies most likely with the Reactis test generator. Reactis tries to cover as many behaviors of the implementation of the system as possible by targeting structural coverage criteria such as decision, or MCDC coverage. These behaviors can then appear in the resulting invariants.

The remainder of this chapter will summarize the contributions and describe how these contributions were achieved followed by a discussion of potential areas for future work.

9.1 Summary of Contributions

We presented two empirical case studies evaluating MBT and one case study that evaluated MBT against manual testing. The studies evaluate the effort and efficiency of the approach to identify ONBs as well as the strength and weaknesses of MBT in general and also with respect to identifying ONBs.

Based on the analysis of the strength and weaknesses of MBT we developed and evaluated an approach that can automatically infer MBT models and their associated test infrastructure from manual test cases for web based systems. The approach was applied to a NASA data analysis system and the tester could successfully apply the approach to identify ONBs that the manual tests missed.

The work presented the Specstractor framework and its associated tool-chain. It was developed to extract and analyze specifications of a system in form of system invariants from automatically generated test cases with the help of data mining. In an evaluation on several systems of the automotive and medical device domain we have demonstrated how the resulting invariants relate to ONBs of the system and how these ONBs manifest in the resulting invariants.

We also presented and evaluated improvements to the Specstractor approach that improved its accuracy, performance and the expressiveness of the resulting invariants. To improve the accuracy and performance we introduced information from static data- and control flow analysis to remove false positives and improve the performance of the test generation and data mining steps. The expressiveness of the rules has been enhanced with the introduction of the Range Miner algorithm that can automatically infer association rules over continuously-valued variables with few false positives.

9.2 Future Work

Like the thesis the discussion of future works is divided into two parts. The first part discusses future work of the state-machine based MBT approach. The second part presents the future work of the Specstractor framework.

9.2.1 MBT

Extending the Model Generator. For the model generator we plan to evaluate if the approach can be extended to non-GUI systems and JUnit test cases to determine the wider applicability of the approach. Since not all systems are as structured as web-applications we are want to assess other ways to guide the model generation. One technique we are are planing to evaluate are based on observations of the runtime state of the system, or an abstract representation thereof during the execution of the test cases [55]. Changes in the state space of the system during the execution of an action in the test case correspond to transitions to a different state in the state-machine model.

Generating off-nominal models from nominal ones. Adding off-nominal inputs and behaviors to a testing model can severely impact the models complexity. This increases the overall effort of the modeling task and all consecutive tasks in the MBT process (e.g. test infrastructure creation, test execution, issue analysis). Moreover the manual identification of inputs/behaviors requires experience and can be error prone. To address this we are planing to create an MBT process were the user models the NB of the system and we then generate an off-nominal model from that nominal model. There are two dimensions that have to be addressed in the creation of the off-nominal mode. Firstly, ONBs triggered by unexpected sequences of events and secondly ONB triggered by unexpected input data. We are planing to evaluate mutation and fuzzing based approaches to automatically create an off-nominal model from an existing nominal model, with respect to the sequences of events. For the data dimension we are planing to create parameterized test cases that enable the variation of the input data of the test cases, so that we can test for different off-nominal data inputs.

9.2.2 Specstractor

Extracting temporal specifications. Temporal specifications are important for describing the ordering of events or the elapsed time between events during the execution of a system. They are often used to specify safety- and liveness properties within a system. The data invariants in this work take the form of association rules that can only describe temporal relationships if one of the observed variables explicitly represented time or a state of the system. Even then it can only relate information between two adjacent time intervals or states. We are planing to use temporal query checking [114] to extract event invariants among the data invariants. Temporal query checking requires rule templates an example of which can be seen in Figure 9.1. The *always preceded by* event invariant template relates two data invariants (a and b). In order for the template to hold for these two data invariants it requires that the data invariant b always precedes data invariant a in the test data. As part of this work we will examine existing event templates [29,115] to evaluate if they relay useful information about a system and its associated specification artifacts.

Additional improvements to the test generator. This work already presented some improvements to the test generator (Chapter 7.1) in the form of

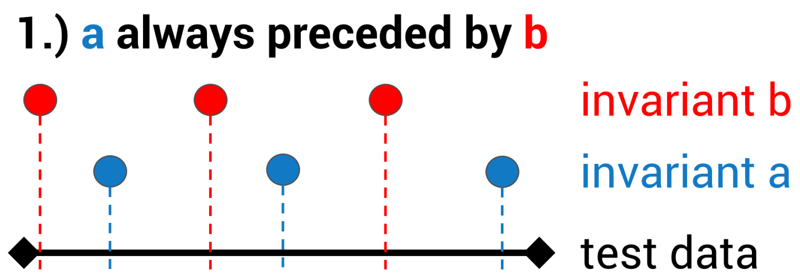


Figure 9.1: Example of an event invariant that relates two data invariants (a and b). The event invariant requires that data invariant b always precedes data invariant a in the test data.

preloading and additional fuzz testing. However, even with these improvements we were not able to remove all false positives from the resulting invariants. We are planing to use model checking tools such as the Simulink Design Verifier [116] to double check the resulting invariants in order evaluate if these tools can remove the remaining false positives.

The test-execution time is one of the dominating factors in the Specstractor approach. The test-execution time is a function of the number of tests to generate and the number of invariants to verify. In order to speed up the test generation process we are planing to parallelize the test execution and perform multiple simultaneous executions over which the invariants are divided.

Improve traceability between invariants and requirements. The process of mapping the resulting invariants against specification artifacts has the downside that any change in the system can change the invariants and require a re-investigation. Establishing traceability links between invariants and the specification would make the approach more robust. If traceability links are in place and the invariants have to be regenerated then the resulting invariants either correspond to already existing invariants and the link will still be intact; if invariants disappear, or change, one could identify broken links.

Bibliography

[1] J. C. Day, K. Donahue, M. Ingham, A. Kadesch, A. K. Kennedy, and E. Post, “Modeling Off-Nominal Behavior in SysML,” jun 2012. [Online]. Available: http://ntrs.nasa.gov/search.jsp?R=20130009268

[2] D. Firesmith. (2012) The need to specify requirements for off-nominal behavior. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2012/01/the-need-to-specify-requirements-for-off-nominal-behavior.html

[3] N. G. Leveson, “Role of software in spacecraft accidents,” *Journal of spacecraft and Rockets*, vol. 41, no. 4, pp. 564–575, 2004.

[4] J.-L. Lions *et al.*, “Ariane 5 flight 501 failure,” 1996.

[5] N. G. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” *Computer*, vol. 26, no. 7, pp. 18–41, July 1993.

[6] J. Day, K. Donahue, M. D. Ingham, A. Kadesch, A. Kennedy, and E. Post, “Modeling off-nominal behavior in sysml.” in *Infotech@ Aerospace*, 2012.

[7] I. Alexander, “Misuse cases: use cases with hostile intent,” *IEEE Software*, vol. 20, no. 1, pp. 58–66, Jan 2003.

[8] T. Ishimatsu, N. G. Leveson, J. P. Thomas, C. H. Fleming, M. Katahira, Y. Miyamoto, R. Ujiie, H. Nakao, and N. Hoshino, “Hazard analysis of complex spacecraft using systems-theoretic process analysis,” *Journal of Spacecraft and Rockets*, 2014.

[9] D. Aceituna, H. Do, and S. Srinivasan, “A systematic approach to transforming system requirements into model checking specifications,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 165–174. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591183

[10] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[11] W. Gilchrist, "Modelling Failure Modes and Effects Analysis," *International Journal of Quality & Reliability Management*,, vol. 10, no. 5, pp. 16–23, may 1996. [Online]. Available: http://www.emeraldinsight.com/doi/10.1108/02656719310040105

[12] D. Aceituna and H. Do, "Addressing the state explosion problem when visualizing off-nominal behaviors in a set of reactive requirements," *Requirements Engineering*, pp. 1–20, sep 2017. [Online]. Available: http://link.springer.com/10.1007/s00766-017-0281-y

[13] B. L. H. David C. Foyle, "Improving evaluation and system design through the use of off-nominal testing: A methodology for scenario development." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.105.9019

[14] M. Schwabacher, J. Samuels, and L. Brownston, "The nasa integrated vehicle health management technology experiment for x-37," 2002.

[15] D. L. Iverson, "Inductive System Health Monitoring With Statistical Metrics," jan 2008. [Online]. Available: https://ntrs.nasa.gov/search.jsp?R=20040068062

[16] N. Kropp, P. Koopman, and D. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*. IEEE Comput. Soc, pp. 230–239. [Online]. Available: http://ieeexplore.ieee.org/document/689474/

[17] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, jun 1975. [Online]. Available: http://ieeexplore.ieee.org/document/6312842/

[18] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard, "Reconciling system requirements and runtime behavior," in *Proceedings of the 9th international workshop on Software specification and design*. IEEE Computer Society, 1998, p. 50.

[19] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Foundations of empirical software engineering: the legacy of Victor R. Basili*, vol. 426, p. 37, 2005.

[20] C. Ackermann, R. Cleaveland, S. Huang, A. Ray, C. Shelton, and E. Latronico, *Automatic Requirement Extraction from Test Cases*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16612-9_1

[21] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000.* IEEE Comput. Soc, 2000, pp. 110–121. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=885865

[22] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[23] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. 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*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/1353673.1353681

[24] A. Kramer, B. Legeard, and R. V. Binder. (2017) MBT User Survey 16/17. [Online]. Available: http://www.cftl.fr/wp-content/uploads/2017/02/2016-MBT-User-Survey-Results.pdf

[25] C. Schulze, M. Lindvall, S. Bjorgvinsson, and R. Wiegand, "Model generation to support model-based testing applied on the nasa dat web-application - an experience report," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 77–87.

[26] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: An empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 135–144. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591180

[27] V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall, and R. Wiegand, "An initial evaluation of model-based testing," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Nov 2013, pp. 13–14.

[28] C. Schulze, D. Ganesan, M. Lindvall, D. M. Omas, and A. Cudmore, "Model-based testing of nasa's osal api; an experience report," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2013, pp. 300–309.

[29] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, ser. SLAML '11.

New York, NY, USA: ACM, 2011, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2038633.2038636

[30] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.01.015

[31] S. Sims and D. C. DuVarney, "Experience report: The reactis validation tool," *SIGPLAN Not.*, vol. 42, no. 9, pp. 137–140, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1291220.1291172

[32] C. Schulze and R. Cleaveland, "Improving invariant mining via static analysis," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 167:1–167:20, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3126504

[33] W. Prenninger and A. Pretschner, "Abstractions for model-based testing," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 59 – 71, 2005, proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104052776

[34] C. Ackermann, A. Ray, R. Cleaveland, J. Heit, C. Martin, and C. Shelton, "Model based design verification: A monitor based approach," in *SAE Technical Paper*. SAE International, 04 2008. [Online]. Available: http://dx.doi.org/10.4271/2008-01-0741

[35] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000. [Online]. Available: http://doi.acm.org/10.1145/335191.335372

[36] N. Leveson, "Systemic factors in software-related spacecraft accidents," *AIAA Space 2001 Conference and Exposition. Paper . . .*, 2001. [Online]. Available: http://arc.aiaa.org/doi/pdf/10.2514/6.2001-4763

[37] J.-C. Fernandez, L. Mounier, and C. Pachon, "A model-based approach for robustness testing," in *TestCom*, vol. 5. Springer, 2005, p. 333.

[38] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007. [Online]. Available: https://books.google.com/books?hl=en{&}lr={&}id=DPAwwn7QDy8C{&}pgis=1

[39] A. K. Ghosh and J. M. Voas, "Inoculating software for survivability," *Communications of the ACM*, vol. 42, no. 7, pp. 38–44, jul 1999. [Online]. Available: http://dl.acm.org/ft{_}gateway.cfm?id=306563{&}type=html

[40] A. Ghosh and M. Schmid, "An approach to testing COTS software for robustness to operating system exceptions and errors," in *Proceedings*

10th International Symposium on Software Reliability Engineering (Cat. No.PR00443). IEEE Comput. Soc, 1999, pp. 166–174. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=809321

[41] "IEEE Standard Glossary of Software Engineering Terminology," pp. 1–84, 1990.

[42] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39–76. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_2

[43] A. Pretschner, S. Wagner, C. K, B. Sostawa, R. Z, and B. M. W. Ag, "One Evaluation of Model-Based Testing and its Automation Categories and Subject Descriptors," no. 3, pp. 392–401, 2005.

[44] L. Apfelbaum and J. Doyle, "Model based testing," in *Software Quality Week Conference*, 1997, pp. 296–300.

[45] H. Reza, K. Ogaard, and A. Malge, "A model based testing technique to test web applications using statecharts," in *Fifth International Conference on Information Technology: New Generations (itng 2008)*, April 2008, pp. 183–188.

[46] A. Heinecke, T. Griebe, V. Gruhn, and H. Flemig, *Business Process-Based Testing of Web Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 603–614. [Online]. Available: https://doi.org/10.1007/978-3-642-20511-8_55

[47] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modeling with fsms," *Software & Systems Modeling*, vol. 4, no. 3, pp. 326–345, Jul 2005. [Online]. Available: https://doi.org/10.1007/s10270-004-0077-7

[48] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55–71, 2011. [Online]. Available: http://dx.doi.org/10.1002/stvr.427

[49] E. Macedo Rodrigues, F. Moreira de Oliveira, L. Teodoro Costa, M. Bernardino, A. F. Zorzo, S. do Rocio Senger Souza, and R. Saad, "An empirical comparison of model-based and capture and replay approaches for performance testing," *Empirical Software Engineering*, oct 2014. [Online]. Available: http://link.springer.com/10.1007/s10664-014-9337-5

[50] N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman, "Achieving both model and code coverage with automated gray-box testing," in *Proceedings of the 3rd international workshop on Advances in model-based testing - A-MOST '07*. New York, New York, USA: ACM Press, jul 2007, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1291535.1291536

[51] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, nov 1987. [Online]. Available: http://dl.acm.org/citation.cfm?id=36888.36889

[52] J. Tretmans, *Model-Based Testing and Some Steps towards Test-Based Modelling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 297–326. [Online]. Available: https://doi.org/10.1007/978-3-642-21455-4_9

[53] A. W. Biermann and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, jun 1972. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5009015

[54] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA: ACM Press, may 2008, p. 501. [Online]. Available: http://dl.acm.org/citation.cfm?id=1368088.1368157

[55] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," in *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, apr 2008, pp. 121–130. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4539539

[56] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014.

[57] A. M. Memon, "An event-flow model of GUI-based applications for testing: Research Articles," *Software Testing, Verification & Reliability*, vol. 17, no. 3, pp. 137–157, sep 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286486.1286487

[58] A. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, oct 2005. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1542069

[59] X. Cheng and M. S. Hsiao, "Simulation-directed invariant mining for software verification," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 682–687. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403541

[60] S. McCamant and M. D. Ernst, "Predicting problems caused by component upgrades," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, p. 287, sep 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=949952.940110

[61] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 683–693. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6227149

[62] A. Gupta and A. Rybalchenko, "InvGen: An efficient invariant generator," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5643 LNCS. Springer, Berlin, Heidelberg, 2009, pp. 634–640. [Online]. Available: http://link.springer.com/10.1007/978-3-642-02658-4{_}48

[63] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[64] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1978, pp. 84–96.

[65] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 288–298, jun 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6227185

[66] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Tacas*, 2005, pp. 461–476. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.115.3535

[67] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009, vol. 5505 LNCS, pp. 292–306.

[68] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*. New York, New York, USA: ACM Press, jul 2010, p. 85. [Online]. Available: http://dl.acm.org/citation.cfm?id=1831708.1831719

[69] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. New York, New York, USA: ACM Press, nov 2007, p. 465. [Online]. Available: http://dl.acm.org/citation.cfm?id=1321631.1321710

[70] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together," in *Automated Software Engineering*, 2012, vol. 19, no. 4, pp. 423–458.

[71] M. Lindvall, C. Ackermann, W. C. Stratton, D. E. Sibol, A. Ray, L. Yonkwa, J. Kresser, S. Godfrey, and J. Knodel, "Using Sequence Diagrams to Detect Communication Problems between Systems," in *2008 IEEE Aerospace Conference*. IEEE, mar 2008, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4526571

[72] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. New York, New York, USA: ACM Press, sep 2011, p. 267. [Online]. Available: http://dl.acm.org/citation.cfm?id=2025113.2025151

[73] I. Dragomir, V. Preoteasa, and S. Tripakis, *Compositional Semantics and Analysis of Hierarchical Block Diagrams*. Cham: Springer International Publishing, 2016, pp. 38–56. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-32582-8_3

[74] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings - International Conference on Software Engineering*. IEEE, 2009, pp. 496–506. [Online]. Available: http://ieeexplore.ieee.org/document/5070548/

[75] A. Agrawal, G. Simon, and G. Karsai, "Semantic translation of simulink/stateflow models to hybrid automata using graph transformations," *Electronic Notes in Theoretical Computer Science*, vol. 109, pp. 43 – 56, 2004, proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104052089

[76] B. Meenakshi, A. Bhatnagar, and S. Roy, "Tool for translating simulink models into input language of a model checker," in *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ser. ICFEM'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 606–620. [Online]. Available: http://dx.doi.org/10.1007/11901433_33

[77] V. Pantelic, S. Postma, M. Lawford, A. Korobkine, B. Mackenzie, J. Ong, and M. Bender, "A toolset for simulink: Improving software engineering practices in development with simulink," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Feb 2015, pp. 1–12.

[78] T. Gerlitz, N. Hansen, C. Dernehl, and S. Kowalewski, "artshop: A continuous integration and quality assessment framework for model-based software artifacts," in *Tagungsband des Dagstuhl-Workshops*, p. 13.

[79] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams: Modularity vs. code size," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: ACM, 2009, pp. 78–89. [Online]. Available: http://doi.acm.org/10.1145/1480881.1480893

[80] R. Lublinerman and S. Tripakis, "Modular code generation from triggered and timed block diagrams," in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2008, pp. 147–158.

[81] ——, "Modularity vs. reusability: Code generation from synchronous block diagrams," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 1504–1509. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403736

[82] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 626–629. [Online]. Available: http://dl.acm.org/citation.cfm?id=1870926.1871074

[83] R. Srikant and R. Agrawal, "Mining quantitative association rules in large relational tables," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/233269.233311

[84] A. Salleb-Aouissi, C. Vrain, and C. Nortet, "Quantminer: A genetic algorithm for mining quantitative association rules." in *IJCAI*, vol. 7, 2007, pp. 1035–1040.

[85] K. Li, C. Reichenbach, Y. Smaragdakis, and M. Young, "Second-order constraints in dynamic invariant inference," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. New York, New York, USA: ACM Press, aug 2013, p. 103. [Online]. Available: http://dl.acm.org/citation.cfm?id=2491411.2491457

[86] R. Sharma and A. Aiken, "From Invariant Checking to Invariant Inference Using Randomized Search," in *Computer Aided Verification*. Springer International Publishing, 2014, pp. 88–105. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9{_}6

[87] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8559 LNCS. Springer, Cham, 2014, pp. 69–87. [Online]. Available: http://link.springer.com/10.1007/978-3-319-08867-9{_}5

[88] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[89] H. Femmer, D. Ganesan, M. Lindvall, and D. McComas, "Detecting inconsistencies in wrappers: A case study," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1022–1031. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486929

[90] V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall, and R. Wiegand, "Model-based testing of nasa's gmsec, a reusable framework for ground system software," *Innovations in Systems and Software Engineering*, vol. 11, no. 3, pp. 217–232, Sep 2015. [Online]. Available: https://doi.org/10.1007/s11334-015-0254-6

[91] V. Gudmundsson, "Model-based testing of flexible systems," 2015.

[92] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[93] D. Ganesan, M. Lindvall, L. Ruley, R. Wiegand, V. Ly, and T. Tsui, "Architectural analysis of systems based on the publisher-subscriber style," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 173–182. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2010.27

[94] D. Ganesan, M. Lindvall, S. Hafsteinsson, R. Cleaveland, S. L. Strege, and W. Moleski, "Experience report: Model-based test automation of a concurrent flight software bus," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2016, pp. 445–454.

[95] Spotify. (2017) Graphwaker. [Online]. Available: http://graphwalker.github.io/

[96] S. J. Prowell, "JUMBL: a tool for model-based statistical testing," pp. 337–345, 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1174916

[97] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*, Sept 2008, pp. 346–355.

[98] yWorks. (2017) yed graph editor. [Online]. Available: http://www.yworks.com/products/yed

[99] O. Source. (2017) Selenium browser automation tool. [Online]. Available: http://www.seleniumhq.org/

[100] Mozilla. (2017) The firefox web-browser. [Online]. Available: https://www.mozilla.org/en-US/firefox/

[101] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993. [Online]. Available: http://doi.acm.org/10.1145/170036.170072

[102] P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam, "The spmf open-source data mining library version 2," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 36–40.

[103] F. Angiulli, G. Ianni, and L. Palopoli, "On the complexity of mining association rules," in *SEBD*, 2001, pp. 177–184.

[104] J. Pei, J. Han, and L. V. S. Lakshmanan, "Mining frequent itemsets with convertible constraints," in *Proceedings 17th International Conference on Data Engineering*, 2001, pp. 433–442.

[105] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang, "Exploratory mining and pruning optimizations of constrained associations rules," *SIGMOD Rec.*, vol. 27, no. 2, pp. 13–24, Jun. 1998. [Online]. Available: http://doi.acm.org/10.1145/276305.276307

[106] L. Hamers, Y. Hemeryck, G. Herweyers, M. Janssen, H. Keters, R. Rousseau, and A. Vanhoutte, "Similarity measures in scientometric research: the jaccard index versus salton's cosine formula," *Information Processing & Management*, vol. 25, no. 3, pp. 315–318, 1989.

[107] Z. Jiang, M. Pajic, A. Connolly, S. Dixit, and R. Mangharam, "Real-time heart model for implantable cardiac device validation and verification," in *2010 22nd Euromicro Conference on Real-Time Systems*, July 2010, pp. 239–248.

[108] P. Fontana, "Towards a unified theory of timed automata," Ph.D. dissertation, University of Maryland, 2015.

[109] S. D. Bay, "Multivariate Discretization for Set Mining," *Knowledge and Information Systems*, vol. 3, no. 4, pp. 491–512, 2001. [Online]. Available: http://link.springer.com/10.1007/PL00011680

[110] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[111] H. Kellerer, U. Pferschy, and D. Pisinger, "Introduction to np-completeness of knapsack problems," in *Knapsack problems*. Springer, 2004, pp. 483–493.

[112] R. Cleaveland, S. A. Smolka, and S. T. Sims, *An Instrumentation-Based Approach to Controller Model Validation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97. [Online]. Available: https://doi.org/10.1007/978-3-540-70930-5_6

[113] M. Lindvall, M. Diep, M. Klein, P. Jones, Y. Zhang, and E. Vasserman, "Safety-focused security requirements elicitation for medical device software," in *Requirements Engineering Conference (RE), 2017 IEEE 25th International*. IEEE, 2017, pp. 134–143.

[114] G. Bruns and P. Godefroid, "Temporal logic query checking," in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 409–. [Online]. Available: http://dl.acm.org/citation.cfm?id=871816.871851

[115] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/302405.302672

[116] J.-F. Etienne, S. Fechter, and E. Juppeaux, "Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain," in *Complex Systems Design & Management*. Springer, 2010, pp. 61–72.