Dissertation

Building Bridges Between Formal Verification and Testing with Automatic Test Generation

Franz Röck[1]

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
A-8010 Graz, Austria

Supervisor/First reviewer:	Prof. Roderick Bloem
Second reviewer:	Prof. Georg Weissenbacher

Graz, February 2018

[1] E-mail: roeck.f@gmx.at

Dissertation

Brücken Bauen zwischen Formaler Verifikation und Testen mittels Automatischer Test Generierung

Franz Röck[1]

Institut für Angewandte Informationsverarbeitung und
Kommunikationstechnologie (IAIK)
Technische Universität Graz
A-8010 Graz, Österreich

Betreuer/1. Gutachter:	Prof. Roderick Bloem
2. Gutachter:	Prof. Georg Weissenbacher

Graz, Februar 2018

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: roeck.f@gmx.at

Abstract

The development process of software usually starts with a big document describing the requirements of the intended product. From this text, which is given in natural language, developers derive the implementation. When formal verification is desired, then it often evolves in a parallel branch in a different domain (formal methods) that aims to prove that the product will satisfy the requirements. Testers, who complement formal methods experts, usually derive their tests from the requirements given in natural language and, thus, do not benefit from the work of this formal branch.

In this thesis we take advantage of the work by formal experts to support the testers. We provide approaches that allow a tester, who is not necessarily an expert in formal methods, to make use of the formalized requirements and formal models and automatically derive tests that can be applied on the real implementation.

We present a technique to generate test cases to test complex Boolean formulas representing for example access policies. We've developed a tool that takes the formula and the desired coverage criterion as an input and calculates variable assignments that test the implementation of the formula. In a case study on the Java Card applet firewall of an industrial implementation we then evaluate our approach.

We've developed another approach that derives test strategies for a given temporal specification to detect specific faults. This approach does not rely on any implementation details and strategies can, thus, be applied to different implementations of the same specification. We evaluate the approach on the AMBA bus arbiter and apply it in a case study on the fault detection isolation and recovery (FDIR) component of a satellite that is currently under development.

Finally, we present a semantics for linear temporal logics (LTL) that allows the user to evaluate LTL properties, which are defined on infinite paths, on finite execution traces which is crucial in testing as all tests are finite. We present an evaluation method that maps inconclusive traces with respect to previously observed behavior, i.e., successful satisfactions of violations, to presumably truth values to support the tester when evaluating many traces. We show that this approach computes also for non-monitorable LTL properties results that match the human intuition.

Keywords: Automatic Test Case Generation, System Testing, Test Strategies, Runtime Verification, LTL.

Kurzfassung

Die Entwicklung von Software startet gewöhnlich mit einem Dokument, welches die Anforderungen an das geplante Produkt enthält. Von diesem Text, der in natürlicher Sprache verfasst ist, leiten die Entwickler in mehreren Schritten die Implementierung ab. Wenn formale Verifikation verlangt wird, dann erfolgt diese zu meist in einem parallelen Prozess, jedoch in einer anderen Domäne (Formale Methoden). Das Ziel der formalen Verifikation ist es zu beweisen, dass das Produkt die Anforderungen erfüllt. Tester hingegen leiten für gewöhnlich die Tests von den - in natürlicher Sprache gegebenen - Anforderungen ab und profitieren daher nicht von der Arbeit der formalen Experten.

In dieser Arbeit entwickeln wir Ansätze, welche es den Testern, die nicht notwendigerweise Experten in formalen Methoden sind, erlauben, sich die formalisierten Anforderungen und formalen Modelle zu Nutze zu machen.

Wir präsentieren eine Technik um Testfälle für komplexe (boolsche) Formeln zu generieren. Wir haben ein Tool entwickelt, welches die Formel und das gewünschte Coverage Kriterium als Input nimmt und entsprechend die Variablenwerte berechnet. An einem industriellen Fallbeispiel, der Java Card applet firewall, evaluieren wir unseren Ansatz.

Wir haben einen weiteren Ansatz entwickelt, der Teststrategien für eine gegebene temporale Spezifikation ableitet um spezifizierte Fehler zu finden. Dieser Ansatz benötigt keine implementierungsspezifischen Details. Die Strategien können daher für jede Implementierung der gegebenen Spezifikation verwendet werden. Wir evaluieren den Ansatz am AMBA bus arbiter und verwenden das Tool zum Testen einer realen Komponente eines Satelliten der sich in Entwicklung befindet.

Schlussendlich präsentieren wir eine Semantik für Lineare Temporale Logik (LTL), die es dem Nutzer und der Nutzerin erlaubt LTL Eigenschaften, die auf unendlichen Pfaden definiert sind, auf endlichen Traces zu evaluieren. Die Evaluierungsmethode bewertet Traces anhand dem bisher beobachteten Verhalten, d.h., hat die Erfüllung einer Eigenschaft schon einmal länger gedauert, dann ist das ein gutes Zeichen und die Eigenschaft wird vermutlich erfüllt werden. Wir zeigen, dass dieser Ansatz auch für nicht beobachtbare LTL Eigenschaften Ergebnisse liefert, die der menschlichen Intuition entsprechen.

Schlagworte: Automatische Testfall Generierung, System Testen, Laufzeit Verifikation, LTL.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
place, date (signature)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

....
Ort, Datum (Unterschrift)

Acknowledgements

I want to thank everyone who supported me in my studies and this thesis, either directly or indirectly.

In particular, I want to thank my parents Franz and Maria, who made this path in my life possible and who are always here for me. Thank you for always believing in me and thank you for your support and all the small things that are overlooked so easily but that are yet so important, such as some encouraging words whenever necessary. Mum and Dad, I love you. ☺

Next, I thank my beloved wife Hannah, who is always on my side. Thank you for your understanding whenever my thoughts went back to topics of this work, although it was our leisure time and we were having a walk in the park. Thank you for always believing in me and thank you for all your support, also in times where I had doubts on ever finishing this thesis and thank you for all the small things that made every day so much easier for me. Hannah, I love you always and forever. ☺

Special thanks also goes to my supervisor Roderick, who supported me with his experience and expertise and who always helped me to stay focused and not drift away from the main research. Thank you for your crucial questions that allowed me to investigate things from different perspectives and, thus, gain new insights. ☺

Finally, I thank my brothers Oliver and Dominik, as well as all my friends and colleagues, who also shared their precious time with me and supported me, even if it was just a coffee talk that helped me to free my mind. ☺

This research has been financially supported by the European Commission through the project IMMORTAL (644905) and by the Austrian Science Fund (FWF) through the project NewP@ss (835917).

Franz Röck
Graz, Austria, February 2018

Danksagung

Ich bedanke mich bei allen Menschen die mich direkt oder auch indirekt bei der Erstellung dieser Arbeit unterstützt haben.

Im speziellen danke ich meinen Eltern Franz und Maria, die mir diesen Weg in meinem Leben ermöglicht haben und immer für mich da sind. Danke, dass ihr immer an mich glaubt und danke für eure Unterstützung, den Rückhalt und alle die kleinen Dinge die so leicht übersehen werden und die doch so wichtig sind, wie etwa ein paar aufmunternde Worte wenn es gerade notwendig ist. Mama und Papa, ich hab euch lieb. ☺

Dann danke ich meiner geliebten Frau Hannah, die immer für mich da ist. Danke für dein Verständnis wenn ich in der Freizeit, etwa bei Spaziergängen, mit meinen Gedanken zu Themen dieser Arbeit abgeschweift bin. Danke, dass du immer an mich glaubst und danke für all deine Unterstützung, etwa wenn ich manchmal an der Fertigstellung dieser Arbeit gezweifelt habe, und danke für all die vielen Dinge die mir jeden Tag erleichtert haben. Hannah, ich liebe dich. ☺

Ein besonderer Dank gilt auch meinem Betreuer Roderick, der mich mit seiner Erfahrung und Expertise unterstützt und mich immer wieder in die richtige Richtung gelenkt hat, wenn ich Gefahr lief zu weit vom Forschungsthema abzukommen. Danke für deine oftmals kritischen Fragen, die es mir ermöglicht haben all die Dinge aus verschiedensten Perspektiven zu betrachten und dadurch neue Erkenntnisse zu gewinnen. ☺

Schlussendlich danke ich meinen Brüdern Oliver und Dominik, sowie all meinen Freunden, Bekannten und Kollegen, die mir ebenfalls wertvolle Zeit ihres Lebens geschenkt und mich unterstützt haben, und sei es nur durch einen Kaffeetratsch, der mir half, meinen Kopf wieder frei zu bekommen. ☺

Diese Arbeit wurde von der Europäische Kommission durch das Projekt IMMORTAL (644905) sowie vom Österreichischen Wissenschaftsfonds (FWF) durch das Projekt NewP@ss (835917) finanziell unterstützt.

Franz Röck
Graz, Österreich, Februar 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> There are these two young fish swimming along and they happen to meet an older fish swimming the other way, who nods at them and says "Morning, boys. How's the water?" And the two young fish swim on for a bit, and then eventually one of them looks over at the other and goes "What the hell is water?"
>
> David Foster Wallace

High quality is very important for products on the market. As perfect software is the ideal, testing [97, 80] plays an important role. In combination with formal verification it can prove correctness [54, 59] and it can also illustrate that the product is faulty with a single run. So we need to find good tests. The art of testing is, therefore, asking the right questions, i.e., due to the infinite number of possible test cases, the challenge is to derive those that will discover the flaws in the system.

Software development starts with requirements from which a specification document on how to meet these requirements, is generated. From this specification, the system is developed and formal verification, if desired [100], evolves in parallel. While verification of the (real) system is achieved by testing, formal verification is achieved by logical proofs [2, 41, 69]. These proofs may consist of a formal model derived from the system specification and logical properties derived from given requirements. The formal model is model

checked to verify if the desired properties hold on it. This proves that the existing formal model satisfies the given requirements. However, it is the (formal) model that satisfies the given requirements and no verdict is given on the real system.

To use the already existing (formal) models and properties for test case generation seems obvious. Moreover, this establishes a link between the formal verification branch and the implementation branch (see Figure 1.1) that complement each other [59].

In [96], Tretmans defines model based testing as a testing approach that derives the test cases from an (abstract) model of desired behavior of the SUT. Since modelling during the development process of the system is often part of the design phase, attention on model based testing increased as well. If the model is a valid representation of the intended system, i.e., expresses exactly the expected behavior of the intended SUT, then the generated tests are also valid, i.e., the verdicts on the conclusion of the test can be given. This is the necessary assumption on which model based testing is built.

Model based testing not only provides the possibility to generate an arbitrary number of test cases from the model, it also introduces a more structured way to the test case generation process. Methods that cover or explore the model to a certain extent can be automated and those automatic methods allow to produce complex test cases a human tester might never have come up with. Moreover, the model works as a test oracle [59] which is not always easy to come up with [11].

Another advantage of model based testing is that it complements formal verification goals. While formal verification focuses on proving that the model of the system satisfies desired properties, model based testing focuses on checking if the real system conforms to the (formal) model. Models that are verified using formal methods are either an abstraction of the real system, or an isolated component. When using these models for system testing, not only the the isolated component, which may have been verified, is evaluated, but the full composition of the real system and its environment are involved as every part may affect the execution of the test. Considering different platforms on which, and different environments in which, the developed system gets executed goes far beyond the possibilities of what can be formally verified.

And as testing is not only interesting on the real system but also in requirements engineering [55], a formalization of the requirements and the application of testing tools can help to idenify flaws already in the design phase.

However, test case generation from formal descriptions requires people

trained in formal methods and testing. A specification provided in temporal logic considers infinite paths while an execution on the System Under Test (SUT) usually terminates at some point. The resulting trace is, therefore, finite and a verdict has to be given on whether this finite trace satisfies the specification that is defined on infinite paths or not [45]. Some approaches put special attention on the end of the trace that is reflected in the specification [16] or restrict themselves to observable sentences [53].

Providing tools that automate the generation of test cases and the oracle shifts the need of people trained in both disciplines to the development phase of the tool. Once the tool is developed and accepted, the number of people familiar with both disciplines is reduced. The main challenge is what techniques to use to derive test cases and oracles from the models.

In this thesis we focus on extending the rich set of model based testing techniques [44]. We focus on propositional formulas that are too large to be tested combinatorial and too complex for a human. We research a technique to derive a test suite from temporal logic properties only, such that the resulting test suite can be applied to any system claiming to implement these properties. Finally, we propose a runtime verification approach that evaluates the resulting finite trace under a given specification in Linear Temporal Logic (LTL).

We visualize in Figure 1.2 how our tools can be integrated in a chain to get from a formal model to a test verdict. Using our proposed test case generation tools a tester can simply take the temporal logic specification of the system under development to derive a good test suite and can use our runtime verification method as a test oracle to evaluate the resulting traces of the system under test. The tester does not need to be an expert in formal methods anymore but can still benefit from the advantages of formal methods such as unambiguous formal specifications of the desired system behavior.

1.1 Boolean Formulas

The models generated in a company during certification processes can be related to security parts of the SUT and specify for example an access policy. Such a policy can be a complex Boolean formula that expresses under which conditions access is allowed or denied. But the policy can as well be a number of events that has to happen in a defined order before a certain event is allowed. It is of big interest to test if the individual transitions and access rules are implemented as modeled (and verified). To achieve this,

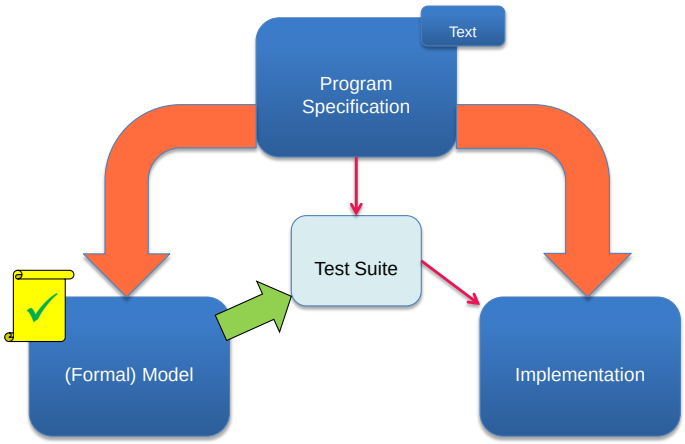


Figure 1.1: Bridging the gap.

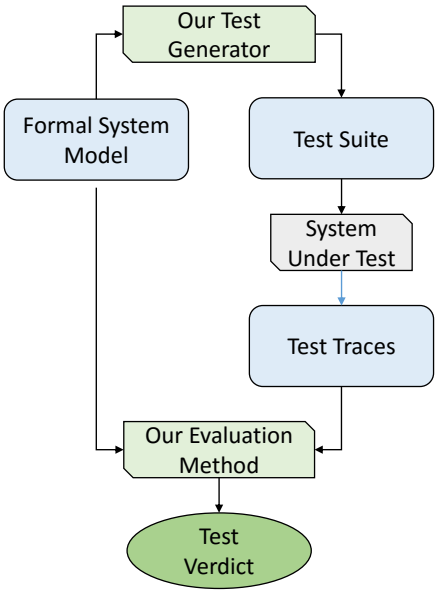


Figure 1.2: From the formal model to a test verdict.

tests need to check if the single parts of a transition guard are implemented and influence the result in the correct way. As these models are verified using a model checker, test case generation using model checker and SAT solver is investigated.

A common approach to generate test cases from a (formal) model is to generate counterexamples using a model checker. A counterexample is a concrete trace through the model. This counterexample is then translated to a concrete test by extracting the inputs and mapping them to inputs of the SUT. Moreover, the trace is as well a test oracle. Outputs extracted from the trace are compared to mapped outputs of the SUT to decide if the concrete test case passed or failed. Tools such as CBMC [40], LLBMC [47], BLAST [22] and Java Pathfinder [56] are only a few of the available software model checking engines.

The idea of generating test cases based on formal specifications was already presented in [19]. Since then, a lot of research has been done in this field [85, 83, 51, 98].

The closest related work we are aware of regarding our test case generation approach is presented in [83]. The authors compute test cases achieving Modified Condition Decision Coverage (MCDC) on a specification by walking through the parse trees of the decisions. Depending on the logical operator they decide what the expression of the subtree should evaluate to. In contrast, our method does not stop at the Boolean level but also produces values for non-Boolean variables appearing in the decisions, and can handle complex dependencies between the idividual parts of the decision. Coverage criteria focusing on graphical representations of a model as well as on transition guards are presented by Ammann et al. in [9].

In [99] the authors define structural coverage metrics for high-level software requirements expressed in LTL. The goal is to exercise the behavior of a system available only as a blackbox. As LTL formulas specify behavior of infinite traces, they discuss structural coverage criteria on these LTL formulas and how they can be adapted to be measured using finite test cases. Finally, they also apply it on a realistic example.

Fraser et al. present a survey of the principles of model-based testing using model checkers [51]. They show that a model checker can be used to generate counterexamples which are translated to test cases. To generate the counterexamples, trap properties are constructed, such as presented in an approach by Beyer et al. in [21]. The resulting test suites meet specified coverage criteria on the model.

A method by Fraser and Ammann [49] ensures that properties are not vacuously satisfied and that faults propagate to observable property viola-

tions (using finite-trace semantics for LTL).

FShell [60] follows this idea using its own query language FQL for specifying which parts of the source code the user wants to cover. The underlying model checker CBMC [40] is used to build a formal representation of the program. Claiming then that the goals specified via FQL cannot be reached, counterexamples are constructed which satisfy the specified coverage goals. This approach is systematic and target oriented, but can be very resource demanding, because the model checkers operate on a formal representation of the *entire* program under test. Also, this approach does not benefit from existing test cases – a scenario which is more common than creating all test cases from scratch. Our test suite augmentation approach in [24] aims at eliminating these shortcomings.

A criterion that tests the influence of every individual condition is the MCDC criterion presented by Chilenski and Miller in [35]. It is required by the US Federal Aviation Administration for safety critical software in aircrafts [92], and also used in other domains like automotive. While generating test suites that satisfy this criterion manually is possible, it soon becomes difficult with increasing size of the formula and coupled conditions. In our work we formalize the MCDC criterion based on informal [35, 34] and semi-formal [8] definitions. Then we automatically generate a test suite that aims for achieving full MCDC coverage on a propositional formula provided by the user. To evaluate our approach, we model the Java Card applet firewall requirements [84] and derive a test suite for the guard that represents the access policy. We then execute the resulting test suite on a Java Card implementation and achieve a code coverage of 89%. We evaluate the outcomes of the individual tests and are able to detect an inconsistency of the specification and the implementation.

1.2 Implementation Independent Tests

Often requirements exist without concrete implementations, like for example protocol or system standards. Sometimes reference implementations exist that can be used as a golden reference but they may still contain yet undetected flaws. While providing explicit input data with expected output data is a way to provide tests, this may not work if there is some implementation freedom that requires the test to react to system specific behavior. Adaptive test strategies are capable of dealing with that and adjust their test behavior to the concrete implementation according to the specification.

To aim for specified faults is the main objective of fault based testing

techniques like mutation testing [65]. Simple faults are introduced into a system implementation or an existing model, then tests are derived that are capable of detecting those introduced faults. Based on two hypotheses, the Coupling Effect [43, 82] and the Competent Programmer Hypothesis [43, 1], the resulting test suite is then also considered to be able to reveal other faults. The Coupling Effect hypothesis states that tests that can detect simple faults are also sensitive to more complex faults and the Competent Programmer Hypothesis states that systems are usually close to a correct version.

Our approach is also a fault based approach and relies on these two hypotheses. While most of the existing work focuses on permanent faults and deterministic system descriptions with unambiguous behavior, we consider transient faults occurring with different frequencies and our approach can uncover faults in every implementation of a given LTL specification (and all behaviors of the uncontrollable part of the system's environment).

To achieve the goal of triggering such a (transient) fault in place of implementation freedom and uncontrollable as well as unspecified parts of the system and the environment, the tests may have to react to observed behavior at runtime and adjust their behavior. Hierons [58] has studied such adaptive test cases from a theoretical perspective, relying on fairness assumptions (every non-deterministic behavior is exhibited when trying often enough) or probabilities. Petrenko et al. compute adaptive tests for trace inclusion [87, 88, 89] or equivalence [86, 75, 89] from a specification given as non-deterministic finite state machine (FSM), also relying on fairness assumptions.

In our method we do not make such assumptions but consider the SUT to be fully antagonistic. Aichernig et al. [3] present a method to compute adaptive tests from (non-deterministic) UML state machines. Starting from an initial state, a trace to a goal state, the state that shall be covered by the resulting test case, is searched for every possible system behavior, issuing inconclusive verdicts only if the goal state is not reachable any more. In contrast, our approach uses reactive synthesis to enforce reaching the desired goal for all implementations if this is possible.

Considering the reactive system to behave antagonistic results in a two player game. Yannakakis points this out in [101], the tester provides inputs with the objective of revealing faults, whereas the SUT provides outputs with the objective of hiding the faults. The tester can only observe outputs and has thus partial information about the SUT. The goal is to find a strategy for the tester that wins against every SUT implementing a given specification. The underlying complexities are studied by Alur et al. in [6].

Our work builds upon reactive synthesis [91] (with partial information [72]), which can also be seen as a game. However, we go far beyond this idea, as we combine the game concept with user-defined fault models. We work out the underlying theory, optimize the faults sensitivity with respect to their frequency, and present a proof-of-concept tool and experimental results for LTL specifications. Nachmanson et al. [81] also synthesize game strategies as tests for non-deterministic software models, but their approach is not fault-based and focuses on simple reachability goals. A variant considers the SUT to behave probabilistically with known probabilities [81]. This model is also used in [23]. Test strategies for reachability goals are also considered by David et al. [42] for timed automata.

Another work that is close to our work is vacuity detection. [17, 73, 10] aim at finding cases where the given specification is trivially satisfied (e.g., because the left side of an implication is false). A good test avoids vacuities in order to challenge the SUT. The method by Beer et al. [17] can produce witnesses that satisfy the specification non-vacuously, which can serve as tests. Tan et al. proposed in [94] a framework for testing LTL properties using the vacuity check [18]. They introduce the property-coverage criterion to generate tests from mutated LTL formulas. Their focus is on generating non-trivial test cases that enable testing the LTL formula on the real system.

Our approach avoids vacuities by requiring that certain faulty SUTs violate the specification. Ammann et al. [7] create tests from CTL [38] specifications using model mutations. This method as well as the previous ones all assume that a deterministic system model is available in addition to the specification.

Fraser and Wotawa [50] also consider non-deterministic models, but issue inconclusive verdicts if the system deviates from the behavior foreseen in the test case. In contrast, when we derive tests with our approach in Chapter 4, we search for test strategies that achieve their goal for *every* realization of the specification.

Boroday et al. [33] aim for a similar guarantee (calling it *strong test cases*) using a model checker, but do not consider adaptive test cases, and use an FSM as a specification.

In this work, we make an assumed fault in the system under test observable. The system claims that it satisfies the requirements and, therefore, has to behave accordingly. Our goal is to force the SUT, from which we assume that it is almost correct and contains only faults we have specified, to violate the requirements assuming the fault is present in the system. To achieve this, we make use of reactive synthesis and generate a strategy for the environment that chooses the inputs to the SUT based on the observed outputs,

Table 1.1: Executions of two different systems that have to implement $\mathsf{GF}p$.

trace		1	2	3	4	5	6	7	8	9	10	11
π_1	p	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–
π_2	p	–	$\top$	–	$\top$	–	–	–	–	–	–	–

such that it forces the SUT to violate the specification if the assumed fault is present. We evaluate our approach on the AMBA protocol [26] and apply the derived test strategies on a concrete AMBA implementation. The strategies are able to not only trigger permanent faults but also transient faults by forcing the system into the assumed fault if it is present in the system. We then apply our approach in a real world case study on the fault detection isolation and recovery component of a satellite.

1.3 Runtime Verification Approach

Whenever tests are derived from a specification, an oracle is necessary that checks whether the resulting finite trace satisfies or violates the specification. However, if the specification is provided in LTL, such as we assume in this thesis, we face the challenge of evaluating a finite trace on properties that are defined on infinite paths. Runtime verification is a lightweight method to check whether the execution of a system satisfies (or violates) given requirements. Properties like "In five steps A has to hold" are easy to monitor, because after five time steps either a satisfaction or a violation of A can be observed. For a property like "Always eventually p has to hold", this is not possible. Because there always exists a continuation that can satisfy the property and there also always exists a continuation that can violate the property. Such properties are simply called non-monitorable. However, there exists work on semantics for interpreting finite traces.

In Table 1.1 we have two traces (of two different systems) that claim to implement the property "Always eventually p has to hold". Manually analyzing trace π_1 gives confidence that the system that produced this trace implements the required property, because $p = \top$ can be observed at every even time step. In contrasts, looking at trace π_2 immediately raises concerns whether the system that produced this trace implemented the property correct, because there is a long suffix with p being false. We use these two traces to highlight the problem with exists finite semantics.

In FLTL [76], the focus is on how to understand $\mathsf{X}(\phi)$. According to

the definition of the next operator in LTL the next state must satisfy the property ϕ. Unfortunately, hitting the end of a finite trace would face the problem that there is no next state anymore. The idea of Manna and Pnueli is to understand the next operator as a strong next operator, i.e., there has to exist a next state that satisfies ϕ, and in addition add a dual operator, the weak next $\overline{\mathsf{X}}$ operator. This operator requires that only if there is a next state, then this state has to satisfy the property ϕ, otherwise it evaluates to true.

A drawback of this proposal is that it requires a rewriting of the specification to decide when a next state shall be considered strong and when it shall be considered weak. And whatever we decide for our example, for both the strong and the weak next operator the verdicts of the traces will not differ from each other for the chosen operator, because they only consider the end of the trace and evaluate it with respect to the chosen operator strength.

Eisner et. al. [46] get around the requirement of rewriting the specification by proposing a weak and a strong view on LTL instead of introducing an additional operator. In the weak view (LTL–) every formula is satisfied by the empty word. In the strong view (LTL+) the empty word does not satisfy any formula.

Still, this semantics faces the same problem as the FLTL semantics. Both traces π_1 and π_2 result in the same outcome when evaluated under the same view.

The authors of [15] approach the problem of finite interpretation for LTL semantics by introducing a three valued semantics. A trace evaluates to $\top$ if it is a good prefix, i.e., for every possible continuation of this trace the property evaluates to true, it evaluates to $\bot$ if it is a bad prefix, i.e., for every possible continuation the property evaluates to false, and it evaluates to ? otherwise.

The problem with this semantics is that the inconclusive evaluation dominates. Also in our example both traces are evaluated to ?, because there always exists a continuation that can satisfy or violate the property.

To refine this inconclusive evaluation, the authors present RV-LTL [16]. This approach combines LTL_3 and FLTL. The resulting semantics is based on the definition of LTL_3, with the inconclusive value ? being replaced with $\top_P$, respectively $\bot_P$, if the trace evaluates to true, respectively false, in FLTL.

Still, this improvement cannot distinguish our two traces.

In [78] the authors highlight cases in which RV-LTL would judge a finite trace to evaluate to presumably false although one would expect that a

continuation of the trace would satisfy the property. One such example is the property $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1) \wedge \mathsf{G}(r_2 \rightarrow \mathsf{F}g_2)$ with alternating requests, i.e., r_1 being high at even time steps and r_2 being high at odd time steps, and every request being granted in the next time step. The authors propose a more detailed refinement for inconclusive traces based on the type of the LTL property. They define for each $\kappa \in \{\mathsf{G}, \mathsf{F}, \text{Prefix}, \mathsf{FG}, \mathsf{GF}, \text{Streett}\}$ a specialized semantics.

Focusing on different classes of temporal logic results in different numbers of truth values for the respective semantics of the class. However, introducing this additional distinctions can still not distinguish between our two given traces, as the approach does also not take observed behavior into account.

All those proposals only focus on the definition of LTL for finite traces. In our proposal we assume that one observes a trace of a finite system that exibits all the good and bad behavior. Our semantics also takes the observed behavior of the system into account when predicting whether an inconclusive trace of this system satisfies or violates a given LTL property. We introduce a distance metric that measures the distance to the satisfaction of the property and another distance that measures the distance to the violation of the property. Based on the collected information we then make predictions on suffixes of the trace that are yet inconclusive. We consider a suffix that is shorter than or equal to the longest sequence it took to satisfy the property to be good, and we consider a suffix that is shorter than or equal to the longest sequence that has violated the property to be bad.

1.4 Thesis Statement

This thesis focuses on automatic test case generation by taking advantage of formal methods. We present an approach to derive tests that investigate individual conditions in a (complex) boolean formula of a system model. Then we present an approach to derive adaptive test strategies that are implementation independent and make a specified fault observable by a specification violation. To conclude whether the system that executes the derived test strategies satisfies the temporal logic specification, we provide a method that automatically evaluates a finite trace with respect to a given LTL specification.

1.5 List of Publications

This thesis is build on the following work:

VALID'13 [27] In this paper we present an automatic test case generation technique for complex boolean formulas. We implemented our approach using an SMT-solver to generate a test suit which achieves MCDC on the formula to cover. Together with Robert Könighofer and Roderick Bloem I formalized the MCDC coverage criterion. I implemented the tool and did the experiments. Under supervision of Karin Greimel at NXP I modeled the Java Card applet firewall used for the case study. The paper was mainly written by me and Robert Könighofer. Karin Greimel contributed the Section about certification and Roderick Bloem proofread the paper. I presented the paper at VALID in Venice in 2013.

TAP'15 [25] This paper presents a case study on automatic test case generation for a secure cache implementation. I developed the formal model of the Secure Block Device. The implementation and testing was done by Richard Schumi in the course of his Master's Thesis under my supervision. I wrote the paper to which Daniel Hein contributed text in the part of the Secure Cache details in the case study section. The paper was proofread by Daniel Hein and Roderick Bloem. I presented the paper at TAP in L'Aquila in 2015.

FMCAD'16 [29] In this paper we present an approach to synthesize adaptive test strategies from temporal logic specifications. The test strategies are implementation independent and aim to reveal a given fault or class of faults. I worked out the idea together with Roderick Bloem, Robert Könighofer and Ingo Pill. Together with Robert Könighofer I worked on the theoretical proofs. I also contributed the optimization for special fault classes. I implemented the approach and evaluated it on AMBA that I've translated into LTL specification. I have written the paper together with Robert Könighofer. Roderick Bloem and Ingo Pill have proofread it. I presented the paper at FMCAD in Mountain View in 2016.

Journal Submission [28]	In this journal paper we extend our work published at FMCAD. We enhance our tool with additional features that compute multiple strategies searching for faults that produce the same failure as specified in the fault model. And we generalize computed strategies to provide additional information to the tester. Together with the German Aerospace Center (DLR) we formalize requirements for a fault detection isolation and recovery unit of a satellite and apply and evaluate our testing approach in this real life scenario. I worked out the enhancements together with Roderick Bloem and I implemented them. I modeled the specification for the new case study and evaluated it together with the people from DLR. I have written the paper together with Heinz Riener.
Conference Submission [14]	In this work we propose a counting semantics for LTL that allows to evaluate LTL properties on finite traces. We first compute for every position of the trace the witness count for satisfaction and the witness count for violation. Based on the computed numbers we predict whether the the respective trace satisfies, violates, presumably satisfies or presumably violates the property or whether the outcome is yet inconclusive. I worked out the core elements of this approach together with Roderick Bloem, I implemented the approach and I was one of the main authors when writing the paper.

Other publications:

QSIC'14 [24]	In this paper we present an approach for automatic test suite augmentation. The implementation takes an existing test suite, evaluates the code coverage and generates new test cases entering previously uncovered code. I designed the idea together with Michael Tautschnig as an enhancement for FShell. I implemented this enhancement with Michael Tautschnig and evaluated it on the Java Card applet firewall. I wrote most of the sections in the paper. Robert Könighofer and Roderick Bloem contributed ideas and proofread the paper. I presented the paper at QSIC in Dallas in 2014.
FDL'16 [4]	This is a joint paper of the IMMORTAL project group. I've provided the section on deriving adaptive test strategies from LTL specifications.

FDL Journal [52]	The FDL paper got extended to become a chapter in **Languages, Design Methods, and Tools for Electronic System Design**.

1.6 Structure of the Thesis

The rest of the thesis is structured as follows. In Chapter 2 we introduce the terminology necessary for this thesis and give some background. This includes informal descriptions of terms and definitions from the field of testing as well as formal definitions from the field of formal methods. In the next three chapters we then present our contributions.

Chapter 3 presents our work of automatically generating test cases for boolean formulas on transition guards in a formal model. We first motivate the test purpose in Section 3.1 and present in Section 3.2 our test case generation approach. Then we present the formal models of our cases studies, the Java Card applet firewall and the Secure Block device, and evaluate our method in Section 3.3.

In Chapter 4 we present our work on synthesizing adaptive test strategies from temporal logic specification. Section 4.1 motivates the test purpose. We present our approach in Section 4.2. Then we give the formal specifications of the toy example and the two case studies we use for the evaluation of our approach, the ARM AMBA bus arbiter and the FDIR component of the Eu:CROPIS satellite, and evaluate our approach in Section 4.3.

In Chapter 5 we present our method to evaluate a finite trace with respect to a given LTL specification. We motivate our method in Section 5.1, before we present in Section 5.2 the counting semantics we've developed to derive more detailed information on the (expected) outcome of the trace. In Section 5.3 we evaluate our method on examples.

Finally, we conclude the thesis and present suggestions for future work in Chapter 6.

Chapter 2

Background

> The most dangerous thing about an academic education is that it enables my tendency to over-intellectualize stuff, to get lost in abstract thinking instead of simply paying attention to whats going on in front of me.
>
> David Foster Wallace

In this chapter we present background knowledge and terminology used in this thesis.

2.1 Terminology

We specify the following terms:

Black-box Testing. In ISO 29119 [62] specification-based testing, which is defined to also be called black-box testing, is a test approach where the knowledge for the generation of the tests *"is the external inputs and outputs of the test item"*.

Dynamic Testing. ISO 29119 [62] defines dynamic testing to be *"testing that requires the execution of the test item"*.

Error. According to IEEE 1044 [30], an error is *"a human action that produces an incorrect result"*. In [67] the author also mentions *mistake* to be a good synonym for an error.

Fault. According to IEEE 1044 [30], a fault is *"a manifestation of an error in software"*. If the error is detected prior to the execution it is called a

defect. In [67] the author also defines a fault as *"[...] the representation of an error, where representation is the mode of expression, such as narrative text, data flow, diagrams, hierarchy charts, source code, and so on"*. The author also further distinguishes between (a) faults of omission and (b) faults of commission. The former defines a fault where we fail to enter correct information, i.e., something is missing but should be present, and the latter defines a fault where we enter something into the representation that is incorrect.

Failure. In IEEE 1044 [30], a failure is also defined as an event in which a function does not stay within specified limits. Hence, a fault has to be executed to result in a failure.

Online Testing. In model based testing, online testing is an approach in which the testing tool is connected directly to the SUT and tests it dynamically, i.e., it can react to the behavior of the SUT and generate inputs on the fly.

Offline Testing. The test suite is created before the tests are executed on the SUT.

System Under Test (SUT). Whenever the test item is the system, we call the test item SUT.

Testing. The goal of testing is to discover faults and demonstrate a correct execution [67]. In ISO 29119 [62] testing is defined to be the *"[...] set of activities conducted to facilitate discovery and/or evaluation of properties of one or more test items"*. It is the investigation of the SUT to collect information. So every action that aims for collecting information on the SUT is essentially a test.

Test Adapter. A test adapter is an interface to the SUT that is capable of executing the test cases and evaluating the results.

Test Case. Both ISO 29119 [62] and [67] define a test case to consist of preconditions, inputs and expected results with the goal of meeting the desired test objectives. The expected results include outputs as well as postconditions that have to hold.

Test Item. In ISO 29119 [62] a test item is the item that is the *"object of testing"*.

Test Oracle. A test oracle determines the correct outputs of the SUT based on inputs.

Test Suite. Is a test set that contains according to ISO 29119 [62] *"one or more test cases with a common constraint on their execution"*.

White-box Testing. In ISO 29119 [62] structure-based testing, which is defined to also be called white-box testing, is a dynamic testing approach where the tests are derived from the internal structures of the test item.

Another term that is often used is glass box testing.

2.2 Logics

While natural language is often ambiguous, we use formal languages to express the intentions. Propositional logic forms the fundamental concept, as it allows the user to rigorously define ones intentions in a formal way. In Section 2.2.1 we introduce the operators for propositional logic that we will need for the definition of the temporal logic LTL, a logic that is capable of specifying propositional behavior over time, in Section 2.2.2. In Section 2.2.3 we then define safety and liveness, the two possible aspects of an arbitrary LTL formula.

A specification of a system as well as a strategy for testing can be expressed as an automaton. Thus, we define automata in Section 2.2.4. In Section 2.2.5 we present model checking and in Section 2.2.6 we present reactive synthesis that we use for our test strategy generation approach.

2.2.1 Propositional Logic

In propositional logic [61], we express atomic sentences that can either be true or false using distinct symbols such as p or q. To define a composition over those atomic sentences we use the following operators:

- $\neg$: The *negation* of an atomic sentence p is denoted by $\neg p$. It expresses the negation of the atomic sentence.
- $\vee$: A composition of two atomic sentences p and q using the *or* operator expresses that at least one of the two statements is true.

Those two operators are enough to express all kinds of compositions, but for better readability there exist abbreviations:

- $\wedge$: A composition of two atomic sentences p and q using the *and* operator expresses that both statements are true. $p \wedge q$ is the abbreviation for $\neg(\neg p \vee \neg q)$.
- $\rightarrow$: A composition of two atomic sentences p and q using the *implication* operator expresses that if the left side is true, then the right side is true as well. $p \rightarrow q$ is the abbreviation for $\neg p \vee q$.
- $\leftrightarrow$: A composition of two atomic sentences p and q using the *equivalence* operator expresses that both sentences have the same evaluation. $p \leftrightarrow q$ is the abbreviation for $(p \rightarrow q) \wedge (q \rightarrow p)$.

2.2.2 Linear Temporal Logic

We use Linear Temporal Logic (LTL) [90] as the specification language for reactive systems. It combines propositional operators ($\neg, \vee$) and temporal operators (X, U).

The syntax of LTL is defined as follows: Every input of the input set I or output of the output set O with $p \in (I \cup O)$ is an LTL formula. The alphabet is $\Sigma = 2^{I \cup O}$ and the set of infinite words over Σ is denoted by Σ^ω. We also refer to words as (execution) traces. If φ_1 and φ_2 are LTL formulas, then $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\mathsf{X}\varphi_1$ and $\varphi_1 \mathsf{U} \varphi_2$ are LTL formulas as well. For an infinite trace $\overline{\sigma} = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ that satisfies LTL formula φ we write $\overline{\sigma} \models \varphi$. An LTL formula φ over an infinite trace $\overline{\sigma}$ is interpreted over the two valued truth domain $\mathbb{B}_2 = \{\top, \bot\}$. We can now inductively define:

- $\overline{\sigma} \models p$ iff $p \in \sigma_0$,
- $\overline{\sigma} \models \neg\varphi$ iff $\overline{\sigma} \not\models \varphi$,
- $\overline{\sigma} \models \varphi_1 \vee \varphi_2$ iff $\overline{\sigma} \models \varphi_1$ or $\overline{\sigma} \models \varphi_2$,
- $\overline{\sigma} \models \mathsf{X}\varphi$ iff $\sigma_1\sigma_2 \ldots \models \varphi$, and
- $\overline{\sigma} \models \varphi_1 \mathsf{U} \varphi_2$ iff $\exists j \geq 0 \,.\, \sigma_j\sigma_{j+1} \ldots \models \varphi_2 \wedge \forall 0 \leq k < j \,.\, \sigma_k\sigma_{k+1} \ldots \models \varphi_1$.

In natural language, $\mathsf{X}\varphi$ requires φ to hold in the next time step and $\varphi_1 \mathsf{U} \varphi_2$ requires φ_1 to hold in every time step until φ_2 holds eventually. In addition LTL also defines $\mathsf{F}\phi$ to be the abbreviation for $\mathsf{true}\mathsf{U}\phi$ and defines $\mathsf{G}\phi$ to be the abbreviation for $\neg\mathsf{F}\neg\phi$.

We extend the definition of traces and also include finite traces. The set of finite words over Σ is denoted by Σ^*. A (finite or infinite) *trace* π is a sequence $\pi_1, \pi_2, \ldots \in \Sigma^* \cup \Sigma^\omega$. We denote by $|\pi| \in \mathbb{N} \cup \{\infty\}$ the *length* of π and we denote by $\pi \cdot \pi'$ the concatenation of $\pi \in \Sigma^*$ and $\pi' \in \Sigma^* \cup \Sigma^\omega$.

As the next operator in standard LTL only allows to explicitly refer to the next time step in the future, we abbreviate a formula of nested next operators that refers to a finite point in the future.

Definition 1 (Nested Next Operators)**.** *Let ϕ be an LTL formula. Then we abbreviate $n \in \mathbb{N}_{>0}$ nested next operators as follows:*
$\mathsf{X}^n\phi \iff \mathsf{X}_1\mathsf{X}_2 \ldots \mathsf{X}_n\phi$.

We now restrict LTL to a fragment with the explicit F operator that we add to the syntax. We provide a 3-valued semantics for this fragment, denoted by $\mu_\pi(\phi, i)$ where $i \in \mathbb{N}_{>0}$ indicates a position in or outside the

trace. We assume the order $\bot <? < \top$, and extend the Boolean operations to the 3-valued domain with the rules $\neg_3\top = \bot$, $\neg_3\bot = \top$ and $\neg_3? =?$ and $\phi_1 \vee_3 \phi_2 = max(\phi_1, \phi_2)$. We define the semantics inductively as follows:

$$
\begin{array}{lcl}
\mu_\pi(p,i) & = & \begin{cases} \top & \text{if } i \leq |\pi| \text{ and } p \in \pi_i, \\ \bot & \text{else if } i \leq |\pi| \text{ and } p \notin \pi_i, \\ ? & \text{otherwise,} \end{cases} \\
\mu_\pi(\neg\phi,i) & = & \neg_3\mu_\pi(\phi,i), \\
\mu_\pi(\phi_1 \vee \phi_2,i) & = & \mu_\pi(\phi_1,i) \vee_3 \mu_\pi(\phi_2,i), \\
\mu_\pi(\mathsf{X}\phi,i) & = & \mu_\pi(\phi,i+1), \\
\mu_\pi(\mathsf{F}\phi,i) & = & \begin{cases} \mu_\pi(\phi,i) \vee_3 \mu_\pi(\mathsf{XF}\phi,i) & \text{if } i \leq |\pi|, \\ \mu_\pi(\phi,i) & \text{if } i > |\pi|, \end{cases} \\
\mu_\pi(\phi_1\mathsf{U}\phi_2,i) & = & \begin{cases} \mu_\pi(\phi_2,i) \vee_3 (\mu_\pi(\phi_1,i) \wedge_3 \mu_\pi(\mathsf{X}(\phi_1\mathsf{U}\phi_2),i)) & \text{if } i \leq |\pi|, \\ \mu_\pi(\phi_2,i) & \text{if } i > |\pi|. \end{cases}
\end{array}
$$

We are aware that this semantics cannot semantically characterize tautologies and contradiction. Evaluating the $p \vee \neg p$ results in ?, although this property is semantically equivalent to $\top$. We decided for this definition as it allows us to evaluate a finite trace in polynomial time. Otherwise we would require a PSPACE-complete algorithm.

In the following lemma, we relate this restricted 3-valued semantics to the standard definition of LTL:

Lemma 2. *Given an LTL formula and a trace $\pi \in \Sigma^*$ with $|\pi| > 0$, we have that*

$$
\begin{array}{rcl}
\mu_\pi(\phi,1) = \top & \Rightarrow & \forall \upsilon \in \Sigma^\omega \,.\, \pi \cdot \upsilon \models \phi, \\
\mu_\pi(\phi,1) = \bot & \Rightarrow & \forall \upsilon \in \Sigma^\omega \,.\, \pi \cdot \upsilon \not\models \phi.
\end{array}
$$

Proof. The proof of these two statements is obtained by induction on the structure of the LTL formula.

□

2.2.3 Safety and Liveness

Alpern and Schneider present in [5] that every property is an intersection of a liveness property and a safety property.

Let ${}^i\upsilon$ denote the finite prefix of the infinite word υ up to the ith position.

A property P is a safety property iff

$$
\forall \upsilon.\upsilon \in \Sigma^\omega.\upsilon \not\models P \implies \exists i.i \geq 0.(\forall \omega.\omega \in \Sigma^\omega.{}^i\upsilon \cdot \omega \not\models P) \qquad (2.1)
$$

Less formal we say that every safety property has a bad prefix. In other words, a safety property can always be violated in finite time.

And a property P is a liveness property iff

$$\forall \nu.\nu \in \Sigma^*.(\exists \omega.\omega \in \Sigma^\omega.\nu \cdot \omega \models P) \tag{2.2}$$

Less formal we say that there always exists a continuation that can satisfy a liveness property. Thus, a liveness property can never be violated in finite time.

2.2.4 Automata

The tester who's part of the environment, as well as the reactive system that is tested, can both be represented as automata. Thus, we specify automata [95].

Let Σ^* denote the set of *finite* words over the finite alphabet Σ and let Σ^ω denote the set of *infinite* words over Σ. Let υ and ν be finite words, then we write $\upsilon \cdot \nu$ to denote the concatenation of those two words. For the concatenation of a finite word υ and an infinite word ω, the resulting word $\upsilon \cdot \omega$ is also an infinite word.

Finite State Machine.

A non-deterministic finite automaton (NFA) is defined as a quintuple $(Q, \Sigma, \Delta, q_0, F)$ where

- Q is a finite, non-empty set of states,
- Σ is the input alphabet consisting of a finite, non-empty set of symbols,
- Δ is a transition function $Q \times \Sigma \to P(Q)$,
- $q_0 \in Q$ is the initial state,
- and F is the set of final (accepting) states, which is a subset of Q.

Let $\alpha = \alpha(0)\alpha(1)\alpha(2)\ldots\alpha(i)$ denote a word over Σ with $\alpha(i) \in \Sigma$ for all $i \in \mathbb{N}$.

A *run* on an NFA is a sequence of states $s = s_0 s_1 s_2 ... s_i$ with $s_i \in Q$, $s_0 = q_0$ and $(s_i, \alpha(i), s_{i+1}) \in \Delta$ for all $i \geq 0$. The automaton accepts a word υ iff for the last state $s_n \in F$ holds. We denote the language $\mathcal{L}(\mathcal{M})$ as the set of strings that is accepted by $\mathcal{M}$.

A deterministic finite automaton (DFA) requires that each transition is unique in its combination of source state and input symbol.

Mealy machines. A Mealy machine is a tuple $\mathcal{S} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \to Q$ is a total transition function that maps a state and an input to a state, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is a total output function that maps a state and an input to an output. For an input trace $\overline{\sigma}_I = x_0 x_1 \ldots \in \Sigma_I^\omega$, $\mathcal{S}$ produces the output trace $\overline{\sigma}_O = \mathcal{S}(\overline{\sigma_I}) = \lambda(q_0, x_0)\lambda(q_1, x_1) \ldots \in \Sigma_O^\omega$, where $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$.

In every time step i, the Mealy machine $\mathcal{S}$ reads the input $x_i \in \Sigma_I$ and maps it together with the current state q_i based on $\lambda(q_i, x_i)$ to an output letter $y_i \in \Sigma_O$. Then it updates its current state q_i according to $\delta(q_i, x_i)$ to the next state q_{i+1}. A Mealy machine allows to model systems with inputs and outputs evolving in discrete time steps.

Moore machines. The Moore machine is a special case of the Mealy machine having $\forall q \in Q \,.\, \forall x, x' \in \Sigma_I \,.\, \lambda(q, x) = \lambda(q, x')$. This states, that the the input x_i does not affect the current output in state q_i as the output function simplifies to $\lambda(q_i)$. It can, however, still affect the next state q_{i+1} and, hence, the output of the next state. We write $\mathsf{Moore}(I, O)$ (resp. $\mathsf{Mealy}(I, O)$) for the set of all Moore (resp. Mealy) machines with inputs I and outputs O.

Reactive Systems. A *reactive system* is a Mealy machine. We say a system *realizes* a specification $L \subseteq \Sigma^\omega$ if $\mathcal{L}(\mathcal{S}) \subseteq L$.

2.2.5 Model Checking

A model checker verifies if a given model satisfies a given specification. Formally, for a given model $\mathcal{M}$ and a given logical property ϕ it is checked if $\mathcal{M} \models \phi$. The concept was first presented in [39]. If the model does not satisfy the specification, a counterexample in form of concrete input values is generated to illustrate a trace that violates the specification. One example of a tool that performs model checking is NuSMV [37]. NuSMVs modeling language allows defining a finite state machine and properties that are checked on the model can be specified using LTL.

In automatic test case generation model checkers are used to generate a test by specifying a **trap property.** A trap property is a term that specifies on purpose a logical property that is not satisfied by the model and, therefore, results in a counterexample. The counterexample is a path through the model that triggers the desired behavior and can be mapped to a test case.

2.2.6 Reactive Synthesis

In this thesis, we use reactive synthesis as a blackbox to automatically compute test strategies from a specification provided in LTL.

The synthesis idea was first presented by Alonzo Church in [36]. The goal of reactive synthesis is to automatically construct a reactive system from a given formal specification [31]. Thus, one only has to provide a list of formalized desired behaviors and the tool computes a model that satisfies those properties.

The two main challenges in reactive synthesis of LTL-properties are that it can become very time consuming as it is 2EXPTIME-complete and that the classical approach is not compositional. These challenges are addressed in three different ways [31]:

- Bounding the size of the desired systems,
- restricting LTL to a subset of its language and the use of specialized algorithms,
- and aiming for partial synthesis which omits the need for complete specifications.

The synthesis procedure requires as input the specification φ given in LTL, the set $I = \{i_1, \ldots, i_m\}$ of Boolean input signals and the set $O = \{o_1, \ldots, o_n\}$ of Boolean output signals.

A realization of the specification φ is a system with the finite set I of Boolean input signals and the finite set O of Boolean output signals. The input alphabet is $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and the alphabet of both is $\Sigma = 2^{I \cup O}$. The set of infinite words over Σ is denoted by Σ^ω.

The synthesis procedure computes, based on the input parameter, either a Moore machine $\mathcal{M} \in \mathsf{Moore}(I, O)$ or a Mealy machine $\mathcal{M} \in \mathsf{Mealy}(I, O)$ that realizes φ. If no system exists that realizes φ, the procedure produces the message unrealizable. We refer to this computation by $\mathcal{M} = \mathsf{synt}(I, O, \varphi)$.

If not all signals are observable, then we require a synthesis procedure with partial information. This synthesis procedure is defined similar to the procedure with complete information that we have presented in the previous paragraph. The difference is that it takes a subset $I' \subseteq I$ of the input signals as additional argument. It computes, again based on the input parameter, a Moore, respectively Mealy, machine $\mathcal{M}' = \mathsf{synt}_p(I, O, \varphi, I')$ with $\mathcal{M}' \in \mathsf{Moore}(I', O)$, respectively $\mathcal{M}' \in \mathsf{Mealy}(I', O)$, that realizes φ while only observing the inputs I'. Again, if no such Moore machine, respectively Mealy machine, exists it produces the message unrealizable.

We also assume that both synthesis procedures, synt and synt_p, take an additional optional parameter Θ, which denotes a set of Moore machines. The respective synthesis procedures compute Moore machines $\mathcal{M} = \mathsf{synt}(I, O, \varphi, \Theta)$ and $\mathcal{M}' = \mathsf{synt}_p(I, O, \varphi, I', \Theta)$ as before with the additional constraint $\mathcal{M}, \mathcal{M}' \notin \Theta$.

We now distinguish a fault from a failure in a Mealy machine. A Mealy machine $\mathcal{S} \in \mathsf{Mealy}(I, O)$ is *faulty* with respect to a given LTL specification φ iff $\mathcal{S} \mid\!\not\models \varphi$, i.e., $\exists \mathcal{M} \in \mathsf{Moore}(O, I) \,.\, \overline{\sigma}(\mathcal{M}, \mathcal{S}) \not\models \varphi$. We call a deviation between the faulty $\mathcal{S}$ and any correct realization $\mathcal{S}'$, i.e., $\mathcal{S}' \mid\!\models \varphi$, a *fault* and we call a trace $\overline{\sigma}(\mathcal{M}, \mathcal{S})$ that reveals the faulty behavior of $\mathcal{S}$, i.e., a trace that violates the given LTL specification φ, a *failure*.

2.3 Testing

This Section gives a general overview on testing and its fundamental concepts of test case identification. Testing can never be complete considering all possible environments. It can, however, discover faults and demonstrate a correct execution [67]. Thus, it is important to rely on hypotheses and aim for a good choice of tests.

In [20], Bernot et al. present the need of test hypotheses and formalize common test practices. In principle they state that if a tester decides on a test suite T and all tests of this test suite pass when executed on the SUT, then he or she assumes that the system is correct for all inputs. This assumption is the result of many, most of the time implicit, assumptions of the tester. The naive hypothesis of having a correct system does not require any test, whereas having no hypotheses at all requires exhaustive testing. Thus, testing is always a tradeoff between hypotheses put on the SUT and the number of tests.

In Section 2.3.1 we present the concept of equivalence classes. This concept is based on the hypothesis that the system (or the component under test) behaves the same for all members of the same equivalence class. In Section 2.3.2 we then present the concept of boundary value testing. This is a technique where the tester goes for the extreme ends of the input values as errors are expected to be more likely observed closer to boundaries. This technique is linked to the equivalence classes in the sense that whenever the tester has to pick elements from such classes he or she may go for values around the boundaries of the respective equivalence class.

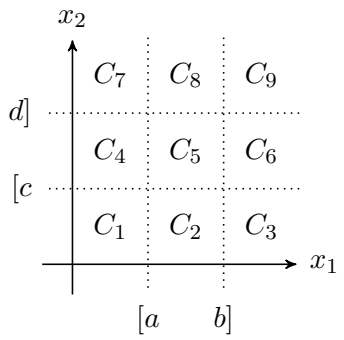


Figure 2.1: Visual representation of the equivalence classes of Equation 2.3.

2.3.1 Equivalence Classes

In Equivalence class testing [67] inputs are grouped into sets and the tester assumes that the SUT behaves identical for all members of the same set. Hence, not every single input value has to be tested anymore, but only elements of different equivalence classes. Example 1 illustrates this concept.

Example 1. Consider a function that takes two inputs $\{x_1, x_2\} \in \mathbb{Z}$ and checks whether both input values are between defined bounds $a, b, c, d \in \mathbb{Z}$:

$$\begin{aligned} a \leq x_1 \leq b \\ c \leq x_2 \leq d \end{aligned} \tag{2.3}$$

In Figure 2.1 we have visualized the nine resulting equivalence classes for Equation 2.3. Every member of equivalence class C_i is assumed to behave the same way as all the other elements in C_i.

Additional Hypotheses have to be put on testing combinations of equivalence classes. The tester has to decide if it is enough to test only one element of every class or if it is necessary to test all possible combinations of elements from (different) classes, or something between. The tester may also merge several equivalence classes based on more hypotheses. In Example 1 all classes but C_5 may for example be merged into one equivalence class assuming that the equivalence classes shall only reflect valid and invalid inputs.

2.3.2 Boundary Value Testing

In boundary value testing [67], the tester assumes that errors are more likely to be observed when the test data is close to the bounds. So the tester

Table 2.1: Boundary value tests for $a \leq x_1 \leq b$.

#	Test	value
1	a^+	$a - \epsilon \leq x_1 < a$
2	a	$x_1 = a$
3	a^-	$a < x_1 \leq a + \epsilon$
4	b^-	$b - \epsilon \leq x_1 < b$
5	b	$x_1 = b$
6	b^+	$b < x_1 \leq b + \epsilon$

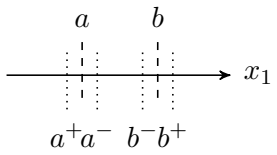


Figure 2.2: Visual representation of the boundary values of Example 2.

basically refines the equivalence classes. Whenever the tester has to pick an element of an equivalence class, he or she may apply different policies for choosing one or more values of the class. Such a selection can be a triplet $< b^+, b, b^- >$, where b^+ is a value that is close to the bound but outside of the class, b is the value that is exactly the bound (within the class) and b^- is a value that is close to the bound and inside the class. The set can also contain only the exact bound or additional values like values that are further away from the bound. In Example 2 we illustrate this approach.

Example 2. Consider a function that takes one input $\{x_1\} \in \mathbb{Z}$ and checks if the input is between defined bounds $a, b \in \mathbb{Z}$:

$$a \leq x_1 < b$$

The tester constructs the equivalence classes $C_1 = \{x_1 < a\}, C_2 = \{a \leq x_1 < b\}$ and $C_3 = \{x_1 \geq b\}$ and identifies the boundary between C_1 and C_2 and the boundary between C_2 and C_3. The tester decides to generate a triplet $< {x_1}^+, x_1, {x_1}^- >$ for these two boundaries. The resulting test classes are presented in Table 2.1 and visualized in Figure 2.2.

2.4 Quality of a Test Suite

The quality of a test suite is difficult to assess. The best test suite would detect all errors. However, as the number of errors and their effect is unknown, there is no way to derive a number for the progress of testing. To accommodate this shortage, measuring the achieved coverage on the source code is a widely used technique. We present existing concepts in Section 2.4.1. In Section 2.4.2 we present a different concept called mutation testing that focuses on identifying which faults in the system the test suite can detect.

2.4.1 Control Flow Criteria

The coverage is measured by executing the test suite and measuring which parts of the source code are executed.

Line coverage. This measure provides a percentage on the number of lines that are executed by the test suite. Let $lines_{total}$ be the total number of lines in the source code and let $lines_{cov}$ be the number of lines covered by the test suite. Then line coverage is calculated as described in Equation 2.4.

$$\text{line coverage} = \frac{lines_{cov}}{lines_{total}} * 100 \tag{2.4}$$

Unfortunately, the resulting percentage heavily depends on the coding style. Imagine you concatenate several strings. If you do this with one line per concatenation you have a different total number of lines, and thus a different percentage of line coverage than concatenating all the strings in one line. This criterion also gives no information on executed branches of conditional statements.

Branch coverage. This measure tells the percentage of executed branches. Let $branches_{total}$ be the number of all branches and let $branches_{cov}$ be the number of branches entered by the test suite. Then branch coverage is calculated as described in Equation 2.5.

$$\text{branch coverage} = \frac{branches_{cov}}{branches_{total}} * 100 \tag{2.5}$$

Whenever an execution reaches a branching point, it evaluates a conditional statement. The coverage of this branching condition can again be subject of coverage analysis. There exists a number of coverage criteria that differs in the requirement on how detailed the conditional statement itself is tested.

Decision Coverage is the least detailed one. It requires the conditional statement only to evaluate once to false and once to true. Let T be the test suite and let d be the branching condition. Let $f(d,t)$ with $t \in T$ be the evaluation of the branching condition d when executing test case t. Then test suite T achieves full Decision Coverage on branching condition d iff

$$\exists t \in T.f(d,t) \wedge \exists t' \in T.\neg f(d,t') \text{ } holds. \tag{2.6}$$

This Coverage criterion only scratches the surface of the branching condition as it does not trigger specific parts of the branching condition.

Condition Coverage requires every single Boolean condition within the conditional statement to evaluate once to true and once to false. Let T again be the test suite and let C_d be the set of all Boolean conditions in branching condition d. Let $f(c,t)$ with $t \in T$ be the evaluation of condition c when executing test case t. Then test suite T achieves full Condition Coverage on branching condition d iff

$$\forall c \in C_d.\Big(\exists t \in T.f(c,t) \wedge \exists t' \in T.\neg f(c,t')\Big) \text{ } holds. \tag{2.7}$$

Achieving full Condition Coverage is, unfortunately, sometimes also possible with only two test cases, one test case in which all conditions evaluate to true and another one in which all evaluate to false. Due to short circuit evaluation, some parts of the conditional statement may still not be triggered.

Multiple Condition Coverage (MCC) is a very detailed coverage criterion. It requires all combinations of evaluations of all Boolean conditions within the conditional statement to be covered, i.e., a conditional statement consisting of n Boolean conditions requires a test suite consisting of 2^n test cases. So MCC may result in enormously large test suites. This includes infeasible evaluations as well as evaluations that are indistinguishable due to short circuit evaluation, e.g., two evaluations of a condition that does not effect the outcome of the conditional statement due to the evaluations of other conditions.

Modified Condition Decision Coverage (MCDC) is a code coverage criterion that keeps the test suite at a manageable size while still testing a conditional statement in detail on the level of each boolean condition. It is used in different domains and required by standards like the DO-178B [92], the standard for safety critical software in aircrafts. The coverage goal according to the Avionics Handbook [93] is that *each condition must be shown to independently affect the outcome of the decision and that the outcome of a decision changes when one condition is changed at a time.*

In [68] the authors highlight the ambiguity of the textual description of MCDC. The original textual definition makes no constraints on the outcome of other boolean conditions in a decision while one condition has to take all possible outcomes. This results in different interpretations. While [34] mentions three different variants, one is basically a combination of the other two, which are masking MCDC and unique cause MCDC. The latter requires the truth values of other conditions to be constant while one toggles between true and false and the former allows them to change. Therefore, unique cause MCDC is a stricter interpretation and masking MCDC is a weaker interpretation. In the literature, the interpretation as used for masking MCDC is also referred to as *Correlated Active Clause Coverage* by the authors in [8]. We formalize this criterion in Section 3.2.

Another issue that arises in MCDC is that conditions may not be independent. One variable may be part of more than one condition and fixing the truth value for one condition might determine the truth values of other conditions as well. So there exist two cases: Either flipping the truth value of one condition always flips the truth value for other conditions (see Example 3), or there exist some variable assignments for which it is possible to flip the truth value of one condition but not the other (see Example 4). The former is called *strongly coupled* [35] and the latter is called *weakly coupled*.

Example 3. Consider two subformulas $\phi_1 = (a > 5)$ and $\phi_2 = (a \leq 5)$. For every possible assignment of variable a the truth value of ϕ_2 is always the opposite of ϕ_1. Such coupled conditions are called strongly coupled.

Example 4. Consider two subformulas $\phi_1 = (a > 5)$ and $\phi_2 = (a < 9)$. By choosing the assignment of variable a properly it is possible to switch the truth value of ϕ_1 without switching the truth value of ϕ_2. We could for example choose $a = 4$, which results in ϕ_1 evaluate to false and ϕ_2 evaluate to true, and then $a = 6$, which makes ϕ_1 evaluate to true while not changing the evaluation of ϕ_2.

2.4.2 Mutation Testing

A different approach to measure the quality of a test suite is mutation testing as presented in [66]. Faults are inserted in the system on purpose. If the test suite is capable of detecting the introduced fault this mutant is said to be “killed”. It is measured how many of the mutants can be killed by the test suite. The mutation score is a value in the range of zero to one. It is calculated as described in Equation 2.8. The higher the mutation score, the

better the test suite. If all mutants are killed by the test suite the mutation score becomes one.

$$MS = \frac{\text{number of killed mutants}}{\text{number of total mutants}} \tag{2.8}$$

One drawback of this approach is that it requires to compile various mutations of the original system and the execution of the whole test suite on all mutated versions. Nevertheless, meaningful answers on what errors the test suite is capable to discover can be given.

Another drawback is that a modification may not necessarily lead to a different behavior. If the mutated system behaves equivalent to the original one then the mutant is said to be an equivalent mutant. For the correct calculation of the mutation score equivalent mutants have to be identified and excluded. Unfortunately detecting an equivalent mutant is a difficult problem by itself.

Chapter 3

Test Case Generation for a Boolean Formula

> Everything in my own
> immediate experience supports
> my deep belief that I am the
> absolute center of the universe,
> the realest, most vivid and
> important person in existence.
>
> David Foster Wallace

This chapter is based on my already published work [27, 25]. References to these papers are not made explicit.

In this chapter, we present our approach on how to automatically derive a test suite that achieves MCDC on a (complex) Boolean formula with strong coupling. To obtain concrete variable assignments we use a Satisfiability Modulo Theories (SMT) solver. This allows the user to compute values for potentially non-Boolean variables in the formula. Moreover, the solver can handle complex interdependencies like "couplings".

We first motivate the test purpose and present our test case generation approach and a formalization of the coverage criterion MCDC. Then we present the formalization of the Java Card applet firewall, a complex access policy, and the Secure Block Device (SBD), a cache control logic without complex guards, that we use to illustrate our approach for automatic test case generation. Finally, we present the experimental results of our two case studies. We derive a test suite from the complex transition guard that describes the access policy of the Java Card applet firewall, evaluate our test suite on a real industrial implementation and discuss the results. For

the case study of the SBD we use an automatic test case generation tool that implements our approach to not only get a satisfying assignment for a single guard, but to get a full input sequence. The test case generation tool generates trap properties for a model checker to derive the test suite satisfying graph coverage criteria like node coverage and edge coverage.

3.1 Test Purpose - Motivation

Models created during the design phase within the software development process are a great source for automatic test case generation using model based testing techniques. Is the model formally verified, the value of the model increases as it is checked to be correct with respect to given requirements and there is a higher confidence when using it as a test oracle. If the model contains transition guards with large Boolean formulas it may be difficult to manually derive test cases that check the details of this formula. Therefore, we focus on automatically deriving a test suite that tests such guards in detail, i.e., the role of every individual term in the formula.

Imagine you have a policy that accepts everyone older than 21 with at least 3 years experience or at least 4 special skills. In a more formal way we can express this policy as follows:

$$((age > 21) \wedge ((exp \geq 3) \vee (skills \geq 4)))$$

Now you want to test if the system executes every single Boolean condition of the policy in a correct way. Test $\tau_1 = (age, experience, skills) = (19, 0, 0)$ may evaluate to false correctly, but you don't know based on which Boolean condition of the policy. To test for example only the age check of the system, you have to provide a test that only fails the age check but satisfies all other Boolean conditions. So test $\tau_2 = (19, 4, 2)$ only evaluates to false if the age check evaluates to false. Test $\tau_3 = (22, 4, 2)$ checks if the system accepts a correct age and not just always evaluates to false regardless of the age.

The same holds for checks of the experience condition. For test $\tau_4 = (22, 4, 4)$ you don't know if it passed because of the experience or because of the skills, while we can use test τ_3 again to also test the experience check.

Thus, to check the individual Boolean conditions of a large formula one has to choose the variable assignment wisely.

3.2 Test Case Generation

To test a transition guard in detail, we've implemented a tool that takes a boolean formula as an input and computes a test suite T that achieves MCDC on it. Our tool supports both variants of MCDC, i.e., masking MCDC and unique cause MCDC. Moreover, it also supports MCC.

The formula that shall be covered must be provided in SMT-LIB2 format [12]. The tool builds a parse tree of the formula and is then capable of replacing single conditions, which are nodes in the parse tree, with either $\top$ or $\bot$. Applying this replacement we can construct, based on Equation 3.1, queries for all conditions of the formula to compute the necessary variable assignments using the SMT-solver Z3 [79] that satisfy the MCDC criterion. For every query that is satisfiable our tool extracts the satisfying assignment of the variables as a pair of test cases t and t'. Before appending the tests to the test suite T the tool checks if one of these tests already exists, to avoid duplicates.

Formalization of MCDC

In Chapter 2.4.1 we have already presented MCDC. In this section we now present a formalization of the informal description.

First, let $V = \{v_1, v_2, \ldots\}$ be the set of variables of domain $\mathbb{D}$ occurring in a decision φ and let φ be a Boolean formula composed of conditions that are subformulas which evaluate to either true or false and do not contain a Boolean operator. A test case is an assignment $t : V \to \mathbb{D}$ of values to all variables in V.

We write $\mathsf{CoN}(\varphi) = \{c_1, c_2, \ldots\}$ for the set of all condition nodes in the parse tree of decision φ, and $\varphi[c|\top]$, respectively $\varphi[c|\bot]$, for the decision φ with condition node $c \in \mathsf{CoN}(\varphi)$ replaced by $\top$, respectively $\bot$. To express the truth value ($\top$ or $\bot$) of decision φ or condition node $c \in \mathsf{CoN}(\varphi)$ under assignment t, we write $\varphi(t)$ or $c(t)$. Let the set $T = \{t_1, t_2, \ldots\}$ be a test suite containing all test cases.

We say that c_i **determines** φ under t, written $\mathbf{det}(c_i, \varphi, t)$, iff $\varphi(t) \neq \varphi[c_i|\neg c_i(t)](t)$, i.e., negating the truth value of condition c_i changes the truth value of φ.

Test suite T achieves *Masking Modified Condition Decision Coverage* [34] on decision φ (see Example 3.1) iff:

$$\begin{gathered}\exists t, t' \in T : \varphi(t) \wedge \neg\varphi(t') \\ \text{and} \\ \forall c \in \mathsf{CoN}(\varphi) : \exists t, t' \in T : \quad c(t) \wedge \neg c(t') \wedge \mathsf{det}(c, \varphi, t) \wedge \mathsf{det}(c, \varphi, t')\end{gathered} \tag{3.1}$$

The decision φ must evaluate to true and to false on some test. Also, every condition node c must evaluate to true and to false while determining the truth value of φ.

Example 5. Assume the Boolean formula $\varphi = (a > 5) \wedge \big((b > 4) \vee (c < 9)\big)$. The set of condition nodes is $\mathsf{CoN}(\varphi) = \{(a > 5), (b > 4), (c < 9)\}$. To compute a test case t and a test case t' for condition node $(a > 5)$ we need to compute variable assignments $t = \{a, b, c\}$ and $t' = \{a', b', c'\}$ that satisfy the following equation:

$$(a > 5) \wedge \neg(a' > 5) \wedge \mathsf{det}((a > 5), \phi, t) \wedge \mathsf{det}((a' > 5), \phi, t')$$

For $\mathsf{det}((a > 5), \phi, t)$ we have

$$(\top \wedge \big((b > 4) \vee (c < 9)\big)) \leftrightarrow \neg((\bot \wedge \big((b > 4) \vee (c < 9)\big)))$$

and for $\mathsf{det}(\neg(a' > 5), \phi, t')$ we have

$$(\bot \wedge \big((b' > 4) \vee (c' < 9)\big)) \leftrightarrow \neg((\top \wedge \big((b' > 4) \vee (c' < 9)\big)))$$

Passing the query to the SMT-solver may produce the following variable assignments:

$$t = \{a = 6, b = 5, c = 0\} \text{ and } t' = \{a' = 5, b' = 5, c' = 9\}.$$

While this definition of MCDC allows conditions to evaluate to different truth values for t and t' as long as they don't change the evaluation of φ (as also illustrated in Example 5), this is not allowed according to the stricter interpretation of the MCDC description. We say that test suite T achieves *unique cause MCDC* [34] on decision φ iff:

$$\begin{gathered}\exists t, t' \in T : \varphi(t) \wedge \neg\varphi(t') \\ \text{and} \\ \forall c \in \mathsf{CoN}(\varphi) : \exists t, t' \in T : \quad c(t) \wedge \neg c(t') \wedge \mathsf{det}(c, \varphi, t) \wedge \mathsf{det}(c, \varphi, t') \wedge \\ \forall c' \in \{\mathsf{CoN}(\varphi) \backslash c\} : c'(t) = c'(t')\end{gathered} \tag{3.2}$$

Note that Equation 3.2 only differs from Equation 3.1 by having the additional restriction $\forall c' \in \{\mathsf{CoN}(\varphi) \backslash c\} : c'(t) = c'(t')$ that requires the

other condition nodes to evaluate to the same truth value under t and t' (see Example 6).

Example 6. Assume again specification $\varphi = (a > 5) \wedge \big((b > 4) \vee (c < 9)\big)$ and compute again a test case t and a test case t' for condition node $(a > 5)$. For unique cause MCDC we require in addition to masking MCDC that all condition nodes $c' \in \{\mathsf{CoN}(\varphi) \backslash c\}$, which are $\{(b > 4), (c < 9)\}$, satisfy $c'(t) = c'(t')$, i.e., that they evaluate to the same truth value under assignments t and t'. Thus, the SMT-solver may provide the following assignments:

$$t = \{a = 6, b = 5, c = 0\} \quad \text{and} \quad t' = \{a' = 5, b' = 5, c' = 0\}$$

3.3 Experimental Results

To evaluate our approach, we've applied our tool on the access policy of the Java Card applet firewall that is a Boolean formula and expresses under which conditions an access is allowed according to the specification. In a next step we have executed the resulting test suite on a real Java Card applet firewall implementation.

In a second case study we've applied our tool for automatic test case generation on a secure cache implementation. This model consists of more states and less complex transition guards.

Before we present the evaluation of the Java Card applet firewall in Section 3.3.2 and the evaluation of the secure cache implementation in Section 3.3.3, we first introduce the formalizations of the respective case studies in Section 3.3.1.

3.3.1 Formal Models

In this section we present the formalization of the Java Card applet firewall access policy and the model of a secure cache implementation. We use these two formalizations to evaluate our test generation approach in the next two sections.

The Java Card Applet Firewall

Whereas in standard Java every applet runs on its own instance of a virtual machine, the Java Card virtual machine must be able to deal with several (independent) applets. The Java Card applet firewall ensures that applets

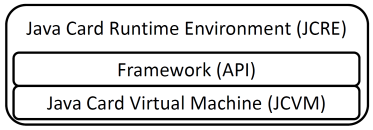


Figure 3.1: The Java Card Runtime Environment.

cannot randomly access data belonging to other applets, but only in restricted cases. The applet firewall is part of the Java Card virtual machine (JCVM) (see Fig. 3.1) and checks every single access according to the JCRE specification [84].

The functionality of the Java Card applet firewall is rather simple. As long as access is granted it idles. If, however, an access is denied, a *SecurityException* is thrown and no more access check is processed until the JCVM processes this exception, e.g. it resets a started transaction and resets the applet firewall.

The complexity for testing lies in the size of the access policy. We want to test if every single defined granted access is also implemented in the system. As the guard that defines when access is granted is a complex decision, deriving tests is not so simple anymore.

Access is granted according to Section 6.2.8 of the Java Card Runtime Environment (JCRE) specification [84], i.e., a satisfying assignment of the guard always corresponds to an access that is granted. If an access does not satisfy this guard, it is denied by definition and the applet firewall throws the exception. This whitelistening ensures that every access that is not explicitly allowed, is denied.

In Example 7 we present the formalization for one part of the access policy, we take a closer look on Section 6.2.8.7 of the JCRE specification [84]. The other requirements are formalized the same way, such that in the end a big complex guard describes under which conditions the applet firewall grants access.

Example 7. Section 6.2.8.7 of the JCRE specification [84] specifies access rules for the bytecode `athrow` by saying:

- "If the object is owned by an applet in the currently active context, access is allowed.

- Otherwise, if the object is designated a Java Card RE Entry Point Object, access is allowed.

- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, access is denied."

This text is formalized as follows:

(bytecode = *athrow*)∧
((Owner = FLAG_CurrentlyActiveContext)∨
(FLAG_entryPointJCREObject)∨
(FLAG_CurrentlyActiveContext = 0))

The first line checks if the bytecode equals *athrow*, the second line checks if the owner is the applet that is currently the active context, the third line checks if the object is a Java Card RE Entry Point object, and the last line check if the JCRE, which is encoded with constant 0, is the currently active context.

Formalizations 3.3 to 3.14 on the next two pages present the full access policy of Section 6.2.8 of the JCRE specification [84] with respect to every bytecode.

$$(bytecode = getstatic)\vee \tag{3.3}$$

$$\Big((bytecode = putstatic) \wedge ((FLAG\_CurrentlyActiveContext = 0) \vee (\neg fieldReferenceType)\vee \tag{3.4}$$
$$(\neg(FLAG\_Val\_entryPointJCREObject \wedge FLAG\_Val\_temporaryJCREObject)\wedge$$
$$\neg FLAG\_Val\_global))\Big)\vee$$

$$\Big((bytecode = aload)\wedge \tag{3.5}$$
$$((FLAG\_CurrentlyActiveContext = 0) \vee (Owner = FLAG\_CurrentlyActiveContext) \vee FLAG\_global)\Big)\vee$$

$$\Big((bytecode = astore)\wedge \tag{3.6}$$
$$((FLAG\_CurrentlyActiveContext = 0) \vee (((\neg fieldReferenceType)\vee$$
$$(\neg(FLAG\_Val\_entryPointJCREObject \wedge FLAG\_Val\_temporaryJCREObject) \wedge \neg FLAG\_Val\_global))\wedge$$
$$((Owner = FLAG\_CurrentlyActiveContext) \vee FLAG\_global)))\Big)\vee$$

$$\Big((bytecode = arraylength)\wedge \tag{3.7}$$
$$((FLAG\_CurrentlyActiveContext = 0) \vee (Owner = FLAG\_CurrentlyActiveContext) \vee FLAG\_global)\Big)\vee$$

$$\Big((bytecode = checkcast)\wedge \tag{3.8}$$
$$((FLAG\_CurrentlyActiveContext = 0) \vee (Owner = FLAG\_CurrentlyActiveContext) \vee FLAG\_global\vee$$
$$FLAG\_entryPointJCREObject \vee (FLAG\_shareableInterfaceObject \wedge exShareable))\Big)\vee$$

$$\Big((bytecode = instanceof)\wedge \tag{3.9}$$
$$((FLAG\_CurrentlyActiveContext = 0) \vee (Owner = FLAG\_CurrentlyActiveContext)\vee$$

$$FLAG\_global \vee FLAG\_entryPointJCREObject \vee (FLAG\_shareableInterfaceObject \wedge exShareable))\Big) \vee$$

$$\Big((bytecode = getfield) \wedge \tag{3.10}$$

$$((FLAG\_CurrentlyActiveContext = 0) \vee (Owner = FLAG\_CurrentlyActiveContext))\Big) \vee$$

$$\Big((bytecode = putfield) \wedge ((FLAG\_CurrentlyActiveContext = 0) \vee (((\neg fieldReferenceType) \vee \tag{3.11}$$

$$(\neg(FLAG\_Val\_entryPointJCREObject \wedge FLAG\_Val\_temporaryJCREObject) \wedge \neg FLAG\_Val\_global)) \wedge$$

$$(Owner = FLAG\_CurrentlyActiveContext)))\Big) \vee$$

$$\Big((bytecode = invokevirtual) \wedge ((Owner = FLAG\_CurrentlyActiveContext) \vee FLAG\_global \vee \tag{3.12}$$

$$FLAG\_entryPointJCREObject \vee (FLAG\_CurrentlyActiveContext = 0))\Big) \vee$$

$$\Big((bytecode = invokeinterface) \wedge \tag{3.13}$$

$$((Owner = FLAG\_CurrentlyActiveContext) \vee FLAG\_entryPointJCREObject \vee$$

$$(FLAG\_CurrentlyActiveContext = 0) \vee (\neg(LCSelectionStatus = NonMultiselectable) \wedge$$

$$((FLAG\_shareableInterfaceObject \wedge exShareable) \vee (FLAG\_CurrentlyActiveContext = 0))))\Big) \vee$$

$$\Big((bytecode = athrow) \wedge ((Owner = FLAG\_CurrentlyActiveContext) \vee \tag{3.14}$$

$$FLAG\_entryPointJCREObject \vee (FLAG\_CurrentlyActiveContext = 0))\Big)$$

The Secure Block Device Cache

In another case study we formalize the SBD, which is a software component written in C for secure persistent data storage [57]. This component does not have complex guards but multiple states instead. We investigate if applying MCDC on rather small guards in a model with multiple states provides an improvement or if there is no gain in code coverage compared to a test suite that only focuses on covering the nodes and edges in the model.

The SBD has to handle management blocks and data blocks. Management blocks that store cryptographic information for a specific number of data blocks are stored in the persistent data storage back-end, where they are interleaved with the blocks containing actual input, the data blocks. To read or write a specific data block, the corresponding management block needs to be in the cache.

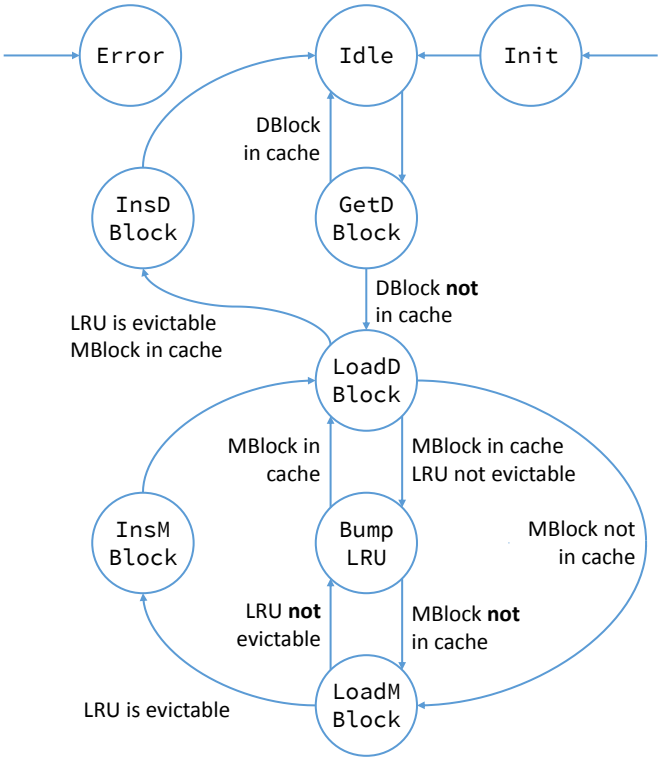


Figure 3.2: Simplified version of the NuSMV model for the cache control logic to access a block.

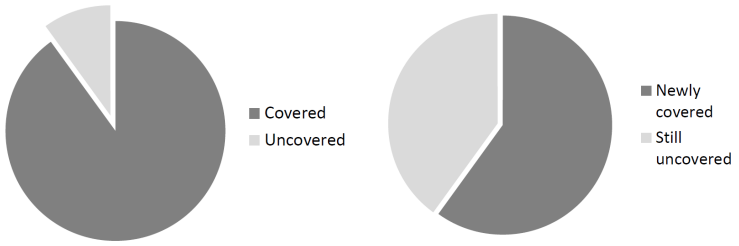
We model the access to a data block (`DBlock`) from the cache. A data

block is either already in the cache, or it has to be loaded, put into the cache and then returned to the caller. A simplified version of the model is illustrated in Figure 3.2.Whenever a data block is requested from the cache, the cache controller first checks if this data block is already in the cache (`GetDBlock`). If so, the block is returned to the caller. If it is **not** in the cache, the cache controller has to check if the corresponding management block (`MBlock`) is in the cache (`LoadDBlock`). If the management block is also **not** in the cache, the cache controller has to load it (`LoadMBlock`) as well. To load the management block the cache controller first evicts the Least Recently Used (LRU) element from the cache. In our analyzed cache it is possible that the LRU element **cannot** be evicted (`BumpLRU`), then it gets pushed to a higher position in the index such that a different block in cache becomes the new LRU element. This happens only if the LRU element is a management block (M_{LRU}) that we do not want to evict. There exist two cases when we do not want to evict the LRU element. We do not want to evict management blocks where there is at least one corresponding data block for M_{LRU} in the cache, and we also do not want to evict the management block that corresponds to the data block that was requested by the caller. In the first case, we bump M_{LRU} until it is more recently used than its most recently used corresponding data block. In the second case, we make it the most recently used element. Once the management block is in the cache, the cache controller loads the data block. Again, the cache controller has to evict the LRU element. If it can be evicted, the cache controller loads the data block and returns it to the caller. If the LRU element **cannot** be evicted the cache controller goes into state `BumpLRU` until a cache slot is available, and then proceeds to load the data block and return it.

3.3.2 Java Card Applet Firewall

To evaluate the test suite derived from the access policy presented in Section 3.3.1, we compare it to the hand-crafted test suite of the Java Card Technology Compatibility Kit (JCTCK). While our test adapter is implemented in C and our test cases only test the applet firewall module, the tests of the JCTCK are provided as Java Card applets and test the full implementation of the Java Card runtime environment.

As the two test approaches are quite different and our test suite only focuses on the access policy of the Java Card applet firewall, we only analyze results with respect to the applet firewall module and measure the achieved code coverage in the corresponding source code related to JCRE specification



(a) JCTCK coverage

(b) Additional coverage on the previously uncovered code

Figure 3.3: Additional coverage on previously from the JCTCK uncovered code achieved by our test suite.

Table 3.1: Instrumentations and achieved coverage

test suite	covered / total	percentage
JCTCK	64/71	90%
our test suite	63/71	89%
together	68/71	96%

Section 6.2.8. Therefore, the source code of the Java Card applet firewall is instrumented by a code coverage tool, such that a test suite that covers all instrumentations is a test suite that achieves condition coverage and basic block coverage. We also point out limitations from the type of test that hinder the test case to achieve the intended goal, e.g. the Java Card applet is restricted when creating an object.

Not all instrumentations are reachable. Some of them are preceded by Exceptions and can never be reached. We manually analyzed the source code and from originally 78 instrumentations only 71 are actually reachable and correspond, therefore, to 100% coverage. We execute both test suites and collect the coverage information. While none of the two test suites was able to achieve full condition and basic block coverage, both achieved a high coverage of approximately 90% (see Table 3.1). Combining the two improved the coverage to 96%. Our test suite was, therefore, able to cover 60% of instrumentations missed by the JCTCK (see Figure 3.3).

Coverage Analysis

As the coverage is high for both test suites, we manually analyze the uncovered code and discuss why it was not reached by the respective test suite. For a summary of uncovered conditions see Table 3.2.

One condition on which our test suite did not cover both possible evaluations is a null pointer check of an object, which in our test suite never evaluated to true. In the implementation, some functions perform such null pointer checks before using the pointer, however, null pointers are not part of the specification from which we derived our test suite and, thus, not in our focus. Therefore, no tests are generated that aim for covering such a condition.

Another condition that is covered by our test suite but not by the JCTCK is a check if an accessed object is a global array. In none of the test applets of the JCTCK the condition evaluated to `true`. Due to the type of test, a Java Card applet, it is not possible for the JCTCK to achieve this, because the incomplete covered condition is disjunct with another condition that checks whether the object is a temporary entry point object. So to cover this condition that checks for an global array, the test needs to generate a global array that is not a temporary entry point object. However, In the Java Card implementation exists only one global array, which is the APDU buffer, being also a temporary entry point object and it is not possible to generate the desired object out of an applet. So the short circuit evaluation, and the lack of other global arrays, makes it impossible to have this global array check evaluate to `true` for the JCTCK. Our test suite is able to cover both truth values of the condition as our test adapter is a C module not underlying the Java Card object generation restrictions and, therefore, capable of generating a global array that is not a temporary entry point object.

Two other conditions are not covered for both possible evaluations by both our test suite and the one from the JCTCK. These conditions check whether an object is a shareable object and whenever evaluated never result in `false`. An analysis of the source code shows that it is impossible to cover the condition evaluating to `true`, because the implementation already performs a check if the class or interface is shareable in the implementation of the function handling the bytecode and otherwise does not call the firewall function at all.

To conclude the coverage analysis, the combination of the two test suites covered every reachable instrumentation. So full condition coverage and basic block coverage with respect to Section 6.2.8 of the JCRE specification is achieved when running both of the test suites.

Table 3.2: Conditions which were not fully covered

condition	JCTCK	our test suite
is the object a null pointer	-	not to true
is the object a global array	not to true	-
is the object a shareable object	not to false	not to false
access of a shareable object	not to false	not to false

Error Detection

When running the test suites and collecting the coverage information, we also analyzed the behavior and compared the outcome to the expected one. While all tests of the JCTCK passed, the outcome of three tests of our test suite did not match the expected one. Manual investigation revealed that two of them have been false positives, where the Java Card applet firewall did not deny access to objects with a certain combination of attributes. While our test adapter allows us to generate every desired type of an object, a Java Card applet is restricted in object generation by the methods provided by the Java Card implementation. Those methods ensure that these objects with attributes that passed the applet firewall although they shouldn't, can not be generated.

The third test case that failed, however, revealed an inconsistency. Whereas one sentence in Section 6.2.8.9 of the JCRE specification states *"Otherwise, if the object is designated a Java Card RE Entry Point Object, access is allowed"*, the implementation denies access and throws a Security Exception. Further investigations, including previous versions of the specification, confirmed that this access rule was not part of version 2.1 [63]. It was introduced in version 2.2 [64], but not implemented in the source code. Because of limitations when creating an object in a Java Card applet, no test of the JCTCK could have tested for this behavior on the system level, and, therefore, the inconsistency remained undiscovered. While this inconsistency cannot occur in practice at the moment because of restrictions in the Java Card object generations, future versions may offer a generation of such an object and then, this inconsistency can produce a failure.

3.3.3 Secure Cache

In a second case study we evaluated our test case generation approach on a detailed model of a secure cache implementation that contains only simple

Table 3.3: Code Coverage.

Coverage criterion	test cases	line coverage	branch coverage
Node	45	87.10%	58.14%
Edge	357	89.52%	59.30%
Edge with MCDC	924	89.52%	59.30%

transition guards. We want to see if having a test suite that satisfies MCDC and is already close to the actual implementation can still provide a gain in code coverage and if a more sophisticated coverage criterion like this is necessary to detect a serious bug that was difficult for a human to find.

To investigate our questions, we derived for the model in Section 3.3.1 a test suite satisfying MCDC on the transition guards. As a comparison we generated, using trap properties, a test suite achieving node coverage, i.e., a test suite that visits every node in the model at least once, and a test suite achieving edge coverage, i.e., a test suite that takes every edge in the model at least once. While the test suite that aims for node coverage contains only 45 test cases, the one achieving edge coverage contains 357 test cases and the test suite aiming for edge coverage with MCDC contains even 924 test cases.

Coverage Analysis

We executed the test suites and measured the achieved line coverage and branch coverage on the source code of the implementation. Table 3.3 presents the results. The test suite that only satisfies node coverage on the model achieves a high line coverage of 87.1% and the more sophisticated test suite which achieves edge coverage on the model increases the line coverage on the source code to 89.53%. The test suite achieving edge coverage with MCDC cannot push the coverage any higher. We observe a similar result with respect to branch coverage. Whereas the test suite satisfying node coverage on the model is capable of covering 58.14% of the branches in the implementation, the other two cover only slightly more branches.

Simple node coverage is in this case study nearly as good as the other two with respect to code coverage. If we look again on the model in Figure 3.2, then we notice that achieving node coverage also requires to cover most of the edges as well. This observation is confirmed by the coverage results in Table 3.3.

In general, one would also expect to achieve an increase in branch cover-

age for the more sophisticated coverage criterion edge coverage with MCDC as it not only generates tests that evaluate the whole guard once to true and once to false, but every single subcondition. If the SUT implements the transition guard by more than one branching condition, then the test suite likely produces a higher branch coverage. In our implementation, however, the transition guards of the model are close to the implemented branching statements and the more sophisticated coverage criterion can, therefore, not improve the branch coverage.

Error Detection

Besides evaluating the coverage of the different test suites on the actual implementation of the SBD, we are also interested in evaluating if the test suites are able to find bugs. To do this, we patched a serious bug from a previous version of the source code into the current version. While the error was hard to find for a human because it is only triggered on a complex control flow, the test suite satisfying node coverage was able to detect this bug. This is due to the high detail level of the model as some of the nodes can only be reached via a certain list of actions. And while a systematic coverage of the detailed model also includes such sequences, this bug that was difficult to find for a human, is detected with a test suite that achieves simple node coverage on the model of the implementation.

Chapter 4

Test Case Generation from Temporal Specification

> ...the most obvious, ubiquitous, important realities are often the ones that are hardest to see and talk about.
>
> David Foster Wallace

This chapter is based on and reuses parts from my already published work [29] and a journal version of this work [28] that is yet under review. References to those papers are not made explicit.

In this chapter, we present a test generation approach for reactive systems that computes system-independent adaptive test strategies. We take formalized requirements provided as temporal logic specification, apply a fault model and synthesize a strategy that enforces a specification violation if a fault that satisfies the fault model is present in the system. The computed strategy is capable of revealing the specified simple fault, like an occasional bit-flip, in every realization of the given requirements. Taking hypotheses from fault-based testing into account, we argue that the resulting strategies can also reveal more complex bugs.

We first discuss the test purpose and illustrate the approach in a motivating example. Then we work out the underlying theory and present the test case generation approach. We apply it on two examples, the amba bus arbiter specification and a PIN locked door specification, to illustrate that the approach can handle industrial sized specifications as well as specifications requiring complex test strategies. Then we apply our approach in a

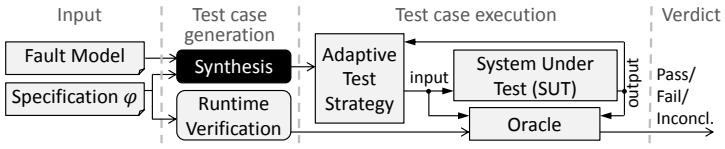


Figure 4.1: Our testing setup.

real world case study on the Fault Detection Isolation and Recovery (FDIR) system for the Eu:CROPIS satellite developed at the German Aerospace Center (DLR).

4.1 Test Purpose - Motivation

Model checking is a valid way to obtain confidence in the correctness of the system. However, it is not always applicable due to components where no code is available, like third-party IP components, or due to scalability issues. Moreover, building a precise model may require high effort and in the end model checking still cannot verify the final and "live" product, but only the (abstracted) model itself.

Testing is a natural choice complementing formal methods. Blackbox techniques do not need any insight of the system and can be generated before the actual system has been implemented. Moreover, as specifications and requirements are usually much simpler than the actual implementation, scalability on this abstraction level is less of a problem. Also, the requirements focus on critical aspects of the intended system and, thus, thorough testing is necessary.

One of the main challenges when deriving tests from the requirements is controllability, because there is plenty of implementation freedom that may result in different system behavior for given inputs. Test cases have to adapt to the system behavior which makes fixed input sequences impossible. Usually, testing approaches solve this issue by requiring a deterministic or probabilistic model of the intended implementation that fixes the behavior in a defined way, which is not necessarily required by the requirements.

Fig. 4.1 presents our assumed testing setup, i.e., how our approach for synthesizing adaptive test strategies (illustrated in black) is integrated in the testing chain. The user provides a specification ϕ that expresses the requirements of the SUT in LTL. Moreover, the user provides a fault model, specified in LTL, that defines the coverage subject, i.e., a class of faults for which the resulting test strategy shall enforce a specification violation if the

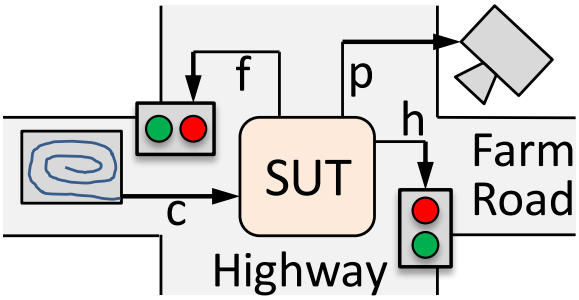


Figure 4.2: Traffic light example.

fault is present in the SUT. We then synthesize test strategies that adapt to the behavior of the SUT, such that the strategies can be executed on any system that claims to implement the given specification. If synthesis is successful, then executing the resulting test strategy long enough guarantees to reveal faults corresponding to the fault model in every realization of the specification. Existing runtime verification methods can be used to derive an oracle from the specification that checks if the SUT conforms to the specification ϕ. Also, in the next chapter we present a semantics to evaluate LTL properties on finite traces.

Imagine a farm road that crosses a highway. At the intersection a traffic light is planned and we have to develop the corresponding controller. There is an induction loop on the farm road that detects if a car is waiting. Figure 4.2 illustrates the crossing. The controller takes as input the Boolean signal c of this detector, being true if a car is idling. Two Boolean outputs of the controller represent the current status of the two traffic lights with h being the signal for the highway and f being the signal for the farm road. The respective signal for a traffic light is true if the light is intended to be green, and false if the light shall be red. In addition, there is a camera that takes a picture whenever a car at the farm road does a jump start, i.e., races off immediately as soon as the traffic light of the farm road turns green. This camera is controlled by the traffic light controller. A Boolean output signal p is true if the camera shall record a picture.

The controller then has to satisfy the following four critical properties:

1. The two lights must never be green at the same time.

2. Whenever a car is waiting at the farm road, the farm road light turns green eventually.

3. Whenever no car is waiting at the farm road, the highway light turns green eventually.
4. The camera shall take a picture, if a car on the farm road does a head start.

We express these critical properties in LTL with

$$\begin{aligned}
\varphi_1 &= \mathsf{G}(\neg f \vee \neg h),\\
\varphi_2 &= \mathsf{G}(c \rightarrow \mathsf{F} f),\\
\varphi_3 &= \mathsf{G}(\neg c \rightarrow \mathsf{F} h),\\
\varphi_4 &= \mathsf{G}\big((\neg f \wedge \mathsf{X}(c \wedge f \wedge \mathsf{X}\neg c)) \leftrightarrow \mathsf{XX} p\big).
\end{aligned}$$

The resulting specification is then:

$$\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$$

A system can implement this specification in many ways. Consider for example the farm road traffic light. A valid implementation of the controller may switch the farm road traffic light to green every once in a while, but it is also valid to switch it to green only if a car is waiting at the crossing. If we have no additional knowledge on implementation details, the only way to test if the farm road's traffic light turns to green is by setting $c = \mathsf{true}$, i.e., relying on Property 2. A correct implementation of Property 2 of the specification requires that the farm road's traffic light turns green ($f = \mathsf{true}$) eventually.

To test whether the camera takes a picture ($p = \mathsf{true}$), a specific input behavior is needed according to the specification. First a car has to be at the crossing and as soon as the farm road's traffic light turns green the car has to start. For this test, a static input sequence is not going to work. The test has to observe the outputs of the system and adapt its behavior accordingly.

A strategy τ_1 that forces the system to set $p = \mathsf{true}$ is illustrated in Figure 4.3. States are labeled by the value of the inputs to the SUT. The transitions between the states are labeled with conditions on observed outputs of the SUT. So all but the first input of the test depend on previous inputs and the correspondingly observed outputs. In the first step, c is set to false. Relying on Property 3 together with Property 1, f has to become false eventually (if the properties are implemented correct). As soon as the adaptive test case observes $f = \mathsf{false}$, c is switched to true, now requiring

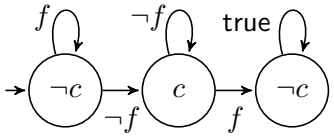


Figure 4.3: Test strategy τ_1 that forces p to be true at least once.

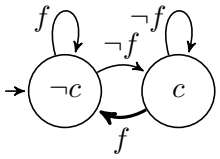


Figure 4.4: Test strategy τ_2 that forces p to be true again and again.

f = true eventually, based on Property 2. When the SUT now switches f to true again, the strategy sets c = false and, therefore, requires the SUT to respond with p = true to satisfy Property 4.

We may also define a class of faults that is not always present. Consider the stuck-at-0 fault at signal p being persistent eventually, but we don't know from which point on exactly. Then our approach may compute strategy τ_2, illustrated in Figure 4.4. This strategy is similar to the previous strategy τ_1, but instead of entering a third state and idling there, the strategy returns to the first state initiating again the sequence that forces the system to switch signal p to true. Thus, if from any time onwards p is stuck-at-0, the system will violate the specification.

Strategies τ_1 and τ_2 are able to reveal a stuck-at-0 fault that manifests permanently at signal p or a stuck-at-0 fault that manifests from some point in time on permanently at signal p, respectively. Let us now assume that a stuck-at-0 fault occurs from some point in time only if a certain input-output interaction happened first, e.g., if c is false at the second time step. Strategy τ_3 as shown in Figure 4.5 guarantees to set c=false in the second time step. The output produced by the SUT responding is not relevant. The strategy then follows τ_2 to enforce p=true infinitely often as before.

The assignment of c to false in the initial state of τ_3 is neither necessary to activate the fault in the envisioned scenario nor to enforce p=true infinitely often. From testing perspective, the tester is free to make an arbitrary choice

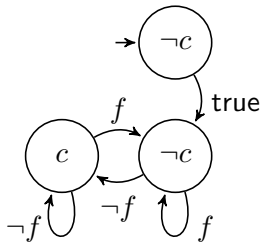


Figure 4.5: Strategy τ_3.

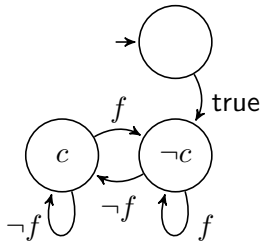


Figure 4.6: Strategy τ_4.

for the input to the SUT in the initial state. As a generalization mechanism of the test strategies, we identify and remove labels from the automaton not necessary to enforce the testing goal. Strategy τ_4 (illustrated in Figure 4.6) is similar to τ_3, but differs by only having assignments for input variables in states where the concrete values are necessary to enforce the desired behavior.

4.2 Test Case Generation

In this section we first take a closer look at the intended coverage objectives. Then we present our approach to derive a test suite of test strategies. We finish this section by presenting extensions to and variants of our approach.

4.2.1 Coverage Objective

In Chapter 2.4 we have presented existing coverage metrics. As the foremost goal of testing is to detect flaws in the SUT, we follow a fault-centered approach and aim for implementations of a faulty model. We assume that the system is "almost correct", i.e., the SUT is a composition of a correct implementation S' according to specification ϕ and a fault F that mutates one of the output signals. This composition is illustrated in Figure 4.7.

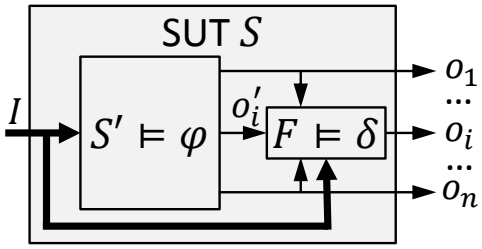


Figure 4.7: Coverage goal illustration.

Our assumption of an "almost correct" system is built on the Competent Programmer Hypothesis [43, 1] that states that implementations are most of the time close to a correct version. As the requirements are on a high abstraction level compared to the actual implementation, a fault model constructed with input and output signals may only model simple faults. Based on the Coupling Effect [43, 82] we argue that strategies that can reveal such simple faults are also sensitive to more complex errors.

Our approach allows the user to define the considered faults in an LTL formula δ. This not only provides the user the possibility to define the type of the mutation, but also when the mutation is expected to be present, i.e., if it's a permanent or transient fault that occurs once or frequently. Examples for such fault models are $\delta_1 = \mathsf{F}(o_i \leftrightarrow o_i')$, which describes a transient bit flip occurring at least once, $\delta_2 = \mathsf{GF}(\neg o_i)$, which expresses a transient stuck-at-0 fault that occurs infinitely often, and $\delta_3 = \mathsf{G}(\mathsf{X}(o_i) \leftrightarrow o_i')$, which models a permanent shift by one time step. In our approach we try to compute a test strategy that can reveal every fault that satisfies δ in every realization of the specification ϕ. We express this coverage objective formally in the following definition.

Definition 3. *A test suite* $TS \subseteq \mathsf{Moore}(O, I)$ *for a system with inputs* I*, outputs* O*, and specification* φ *is universally complete*[1] *with respect to a given fault model* δ *iff*

$$\forall o_i \in O \,.\, \forall \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\} \setminus \{o_i\}) \,.\, \forall F \in \mathsf{Mealy}(I \cup O \cup \{o_i'\} \setminus \{o_i\}, \{o_i\}) \,.\, \exists \tau \in TS \,.\, \Big(\big(\mathcal{S}' \,|\!\models \varphi[o_i \leftarrow o_i'] \wedge F \,|\!\models \delta \big) \rightarrow \big(\overline{\sigma}(\tau, \mathcal{S}' \circ F) \not\models \varphi \big) \Big). \quad (4.1)$$

[1] The word "universal" indicates that the faults are revealed in every (otherwise correct) system.

So, for every output o_i the test suite TS must contain test strategies that enforce a specification violation in every system $\mathcal{S} \mid\!\models \varphi[o_i \leftarrow o_i']$ that contains the fault $F \mid\!\models \delta$ (see Figure 4.7). Note that signal o_i' does not exist in the real system implementation and, thus, cannot be observed by the test strategies $\tau \in \text{TS} \subseteq \mathsf{Moore}(O, I)$. We only introduced this signal to define our coverage objective.

The number of systems that realize $\mathcal{S}' \mid\!\models \varphi[o_i \leftarrow o_i']$ and faults $F \mid\!\models \delta$ can be infinite. Hence, computing a separate test case for every realization is impossible. It's more efficient to have one strategy for every o_i that covers all.

Theorem 4. *A universally complete test suite TS $\subseteq \mathsf{Moore}(O, I)$ with respect to fault model δ exists for a system with inputs I, outputs O, and specification φ if*

$$\forall o_i \in O \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,. \\ \overline{\sigma}(\tau, \mathcal{S}) \models \big((\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi\big). \quad (4.2)$$

Proof. Equation 4.2 implies

$$\forall o_i \in O \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,. \\ \big(\mathcal{S} \mid\!\models \varphi[o_i \leftarrow o_i'] \wedge \delta\big) \rightarrow \big(\overline{\sigma}(\tau, \mathcal{S}) \not\models \varphi\big) \quad (4.3)$$

because (a) going from $\exists\tau\forall\mathcal{S}$ to $\forall\mathcal{S}\exists\tau$ can only make the formula weaker, and (b) $\mathcal{S} \mid\!\models \varphi[o_i \leftarrow o_i'] \wedge \delta$ implies $\overline{\sigma}(\tau, \mathcal{S}) \models (\varphi[o_i \leftarrow o_i'] \wedge \delta)$ for all τ, which can only make the left side of the implication stronger. In turn, Equation 4.3 is equivalent to

$$\forall o_i \in O \,.\, \forall \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\} \setminus \{o_i\}) \,. \\ \forall F \in \mathsf{Mealy}(I \cup O \cup \{o_i'\} \setminus \{o_i\}, \{o_i\}) \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,. \\ \big(\mathcal{S}' \mid\!\models \varphi[o_i \leftarrow o_i'] \wedge F \mid\!\models \delta\big) \rightarrow \big(\overline{\sigma}(\tau, \mathcal{S}' \circ F) \not\models \varphi\big). \quad (4.4)$$

because for a given $\mathcal{S}' \mid\!\models \varphi[o_i \leftarrow o_i']$ and $F \mid\!\models \delta$ from Equation 4.4 we can define an equivalent system $\mathcal{S} = (\mathcal{S}' \circ F) \in \mathsf{Mealy}(I, O \cup \{o_i'\})$ for Equation 4.3 such that $\mathcal{S} \mid\!\models \varphi[o_i \leftarrow o_i'] \wedge \delta$ is satisfied. Also, for a given $\mathcal{S} \mid\!\models \varphi[o_i \leftarrow o_i'] \wedge \delta$ from Equation 4.3 we can define a corresponding $\mathcal{S}' \mid\!\models \varphi[o_i \leftarrow o_i']$ and $F \mid\!\models \delta$ by stripping off different outputs. □

While Equation 4.2 is a sufficient condition for a universally complete test suite to exist, it is not a necessary condition. If it were, computing

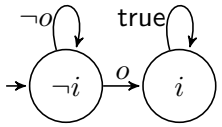


Figure 4.8: Strategy τ_3.

one test strategy per o_i would be enough. Unfortunately, it isn't, as the following example illustrates.

Example 8. Consider a system with input $I = \{i\}$, output $O = \{o\}$, and specification $\varphi = \big(\mathsf{G}(i \rightarrow \mathsf{G}i) \wedge \mathsf{F}i\big) \rightarrow \big(\mathsf{G}(o \rightarrow \mathsf{G}o) \wedge \mathsf{F}o \wedge \mathsf{G}(i \vee \neg o)\big)$. The left side of the implication assumes that the input signal i is set to true at some point and then remains true forever. The right side of the implication requires the same for the output signal o. Moreover, output signal o must not be raised before the input signal i. This specification can for example be realized by a system that always copies the input to the output, i.e., by setting $o = i$.

The test suite $\mathrm{TS} = \{\tau_3\}$ with τ_3 shown in Figure 4.8 is universally complete with respect to fault model $\delta = \mathsf{F}(o \leftrightarrow \neg o')$, which models a bit flip that happens at least once. As long as the input signal i is false, any correct system implementation $\mathcal{S}' \in \mathsf{Mealy}(\{i\}, \{o'\}) \,|\!\models \varphi[o_i \leftarrow o_i']$ must set the output $o' = \mathsf{false}$. Eventually, because of $F \,|\!\models \delta$, the output must flip o to true. At that time the input i is set to true by τ_3 so that the resulting trace $\overline{\sigma}(\tau, \mathcal{S}' \circ F)$ violates φ. Still, Equation 4.2 is false. A closer look at strategy τ_3 confirms that it does not satisfy Equation 4.2. Consider a system $\mathcal{S} \in \mathsf{Mealy}(\{i\}, \{o, o'\})$ that sets $o' = \mathsf{true}$ and $o = \mathsf{false}$ in all steps, then we have $\overline{\sigma}(\tau_3, \mathcal{S}) \models \big(\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi\big)$. So i stays false, and $\varphi[o_i \leftarrow o_i']$ and φ are vacuously satisfied by $\overline{\sigma}(\tau_3, \mathcal{S})$. The fault formula δ is satisfied because $o \leftrightarrow \neg o'$ holds all time. Hence, $\mathcal{S}$ is a counterexample to τ_3 that satisfies Equation 4.2. Similar counterstrategies exist for all other test strategies.

While Equation 4.2 is only a sufficient but not a necessary condition in case of partial observability, it is both sufficient and necessary if all output signals are observable.

The following two lemmas state that (a) the quantifiers can be swapped and (b) the assumption $\overline{\sigma}(\tau, \mathcal{S}) \models A$ is equivalent to the assumption $(\mathcal{S} \,|\!\models A)$ if τ has full information on the outputs in $\mathcal{S}$. Based on these two lemmas we then show that Equation 4.2 is both a necessary and a sufficient condition for a universally complete test suite to exist whenever all output signals are

observable.

Lemma 5. *For every LTL specification ψ over inputs I and outputs O, $\exists\tau \in \mathsf{Moore}(O,I)\,.\,\forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \psi$ holds if and only if $\forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\exists\tau \in \mathsf{Moore}(O,I)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \psi$ holds.*

Proof. Synthesis from LTL specifications under complete information is (finite memory) determined [77], i.e., either

$\exists\tau \in \mathsf{Moore}(O,I)\,.\,\forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \psi$

or

$\exists\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\forall\tau \in \mathsf{Moore}(O,I)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \neg\psi$

holds, but not both. Less formal we can say that either there exists a test strategy τ that satisfies ψ for all systems $\mathcal{S}$, or there exists a system $\mathcal{S}$ that can violate ψ for all test strategies τ. From that, it follows that

$$
\begin{aligned}
& \exists\tau \in \mathsf{Moore}(O,I)\,.\,\forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \psi \\
\text{iff}\quad & \neg\exists\mathcal{S} \in \mathsf{Mealy}(I,O)\,. \\
& \forall\tau \in \mathsf{Moore}(O,I)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \neg\psi \\
\text{iff}\quad & \forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\exists\tau \in \mathsf{Moore}(O,I)\,.\,\overline{\sigma}(\tau,\mathcal{S}) \models \psi.
\end{aligned}
$$

□

Lemma 6. *For all LTL specifications A, G over inputs I and outputs O, we have*

$$
\begin{aligned}
& \forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\exists\tau \in \mathsf{Moore}(O,I)\,. \\
& (\mathcal{S} \mid\!\models A) \rightarrow \big(\overline{\sigma}(\tau,\mathcal{S}) \models G\big) && (4.5) \\
\textit{iff}\quad & \forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\exists\tau \in \mathsf{Moore}(O,I)\,. \\
& \overline{\sigma}(\tau,\mathcal{S}) \models (A \rightarrow G). && (4.6)
\end{aligned}
$$

Proof. Direction $\Rightarrow$: We show that Equation 4.6 being false contradicts with Equation 4.5 being true.

$$
\begin{aligned}
& \neg\forall\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\exists\tau \in \mathsf{Moore}(O,I)\,. \\
& \overline{\sigma}(\tau,\mathcal{S}) \models (A \rightarrow G) \\
\text{iff}\quad & \exists\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\forall\tau \in \mathsf{Moore}(O,I)\,. \\
& \overline{\sigma}(\tau,\mathcal{S}) \models (A \wedge \neg G) \\
\text{iff}\quad & \exists\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\mathcal{S} \mid\!\models (A \wedge \neg G), \text{ which implies} \\
& \exists\mathcal{S} \in \mathsf{Mealy}(I,O)\,.\,\forall\tau \in \mathsf{Moore}(O,I)\,. \\
& (\mathcal{S} \mid\!\models A) \wedge \big(\overline{\sigma}(\tau,\mathcal{S}) \models \neg G\big).
\end{aligned}
$$

Direction $\Leftarrow$: Using the LTL semantics, we can rewrite $\overline{\sigma}(\tau, \mathcal{S}) \models (A \rightarrow G)$ in Equation 4.6 as $\big(\overline{\sigma}(\tau, \mathcal{S}) \models A\big) \rightarrow \big(\overline{\sigma}(\tau, \mathcal{S}) \models G\big)$. Since $\mathcal{S} \mid\!\models A$ implies $\overline{\sigma}(\tau', \mathcal{S}) \models A$ for every $\tau' \in \mathsf{Moore}(I, O)$, the assumption in Equation 4.5 is not weaker, so Equation 4.5 is not stronger. □

For all cases in which all output signals are observable, we use Lemma 5 and Lemma 6 to prove that Equation 4.2 of Theorem 4 is both a necessary and a sufficient condition for a universally complete test suite to exist.

Proposition 7. *Given a fault model of the form $\delta = \mathsf{G}(o_i' \leftrightarrow \psi)$, where ψ is an LTL formula over I and O, a universally complete test suite $TS \subseteq \mathsf{Moore}(O, I)$ with respect to δ, I, O, and φ exists if and only if Equation 4.2 holds.*

Proof. $\varphi[o_i \leftarrow o_i'] \wedge \mathsf{G}(o_i' \leftrightarrow \psi)$ is equivalent to $\varphi[o_i \leftarrow \psi] \wedge \mathsf{G}(o_i' \leftrightarrow \psi)$. Thus, Equation 4.2 becomes

$$\forall o_i \in O \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\, \overline{\sigma}(\tau, \mathcal{S}) \models \big((\varphi[o_i \leftarrow \psi] \wedge \mathsf{G}(o_i' \leftrightarrow \psi)) \rightarrow \neg\varphi\big),$$

which is equivalent to

$$\forall o_i \in O \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \overline{\sigma}(\tau, \mathcal{S}) \models \big(\varphi[o_i \leftarrow \psi] \rightarrow \neg\varphi\big)$$

because of the G operator, a unique value for o_i' exist is all steps and thus, o_i' is just an abbreviation for ψ. Whether this abbreviation o_i' is available as output of $\mathcal{S}$ or not is irrelevant, because τ cannot observe o_i' anyway. Since o_i' no longer occurs, Lemma 5 and Lemma 6 can be applied to prove equivalence between Equation 4.2 and

$$\forall o_i \in O \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,.\, (\mathcal{S} \mid\!\models \varphi[o_i \leftarrow \psi]) \rightarrow \overline{\sigma}(\tau, \mathcal{S}) \not\models \varphi.$$

As τ cannot observe o_i', it is irrelevant whether the truth value of ψ is available as additional output o_i' of $\mathcal{S}$ or not. Hence, the above formula is equivalent to

$$\forall o_i \in O \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,.\, (\mathcal{S} \mid\!\models (\varphi[o_i \leftarrow \psi] \wedge \mathsf{G}(o_i' \leftrightarrow \psi)) \rightarrow \overline{\sigma}(\tau, \mathcal{S}) \not\models \varphi$$

and

$$\forall o_i \in O \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,. \\ (\mathcal{S} \,|\!\models (\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \overline{\sigma}(\tau, \mathcal{S}) \not\models \varphi,$$

i.e., to Equation 4.3. The remaining steps can be taken from the proof of Theorem 4. □

So, if we can rewrite $\varphi[o_i \leftarrow o_i']$ to $\varphi[o_i \leftarrow \psi]$ in Equation 4.2, then the hidden signal is eliminated and it is not only a sufficient condition anymore, but also becomes a necessary one.

Proposition 8. *If the fault model δ does not reference o_i', a universally complete test suite $TS \subseteq \mathsf{Moore}(O, I)$ with respect to δ, I, O, and φ exists if and only if Equation 4.2 holds.*

Proof. We show that Equation 4.2 holds if and only if Equation 4.3 holds. The remaining steps have already been proven for Theorem 4.

Lemma 9. *Equation 4.2 holds if and only if*

$$\forall o_i \in O \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,.\, \forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,. \\ \overline{\sigma}(\tau, \mathcal{S}) \models (\delta \rightarrow \neg\varphi). \tag{4.7}$$

Direction $\Leftarrow$ is obvious because Equation 4.2 contains stronger assumptions (and $\forall \mathcal{S} \in \mathsf{Mealy}(I, O)$ can be changed to $\forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\})$ in Equation 4.7 because $\delta \rightarrow \neg\varphi$ does not contain o_i').

Direction $\Rightarrow$: We show that Equation 4.7 being false *contradicts with Equation 4.2 being* true*.*

$$\begin{aligned}
& \neg\forall o_i \in O \,.\, \exists \tau \in \mathsf{Moore}(O, I) \,. \\
& \forall \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \overline{\sigma}(\tau, \mathcal{S}) \models (\delta \rightarrow \neg\varphi) && (4.8) \\
\textit{iff}\quad & \exists o_i \in O \,.\, \forall \tau \in \mathsf{Moore}(O, I) \,. \\
& \exists \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \overline{\sigma}(\tau, \mathcal{S}) \models (\delta \wedge \varphi) && (4.9) \\
\textit{iff}\quad & \exists o_i \in O \,.\, \exists \mathcal{S} \in \mathsf{Mealy}(I, O) \,. \\
& \forall \tau \in \mathsf{Moore}(O, I) \,.\, \overline{\sigma}(\tau, \mathcal{S}) \models (\delta \wedge \varphi) && (4.10) \\
\textit{iff}\quad & \exists o_i \in O \,.\, \exists \mathcal{S} \in \mathsf{Mealy}(I, O) \,.\, \mathcal{S} \,|\!\models (\delta \wedge \varphi) && (4.11) \\
\textit{iff}\quad & \exists o_i \in O \,.\, \exists \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,. \\
& \mathcal{S}' \,|\!\models (\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi), && (4.12)
\end{aligned}$$

$$
\begin{aligned}
\textit{iff}\quad & \exists o_i \in O \,.\, \exists \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,. \\
& \forall \tau \in \mathsf{Moore}(O \cup \{o_i'\}, I) \,. \\
& \overline{\sigma}(\tau, \mathcal{S}) \models (\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi), && (4.13) \\
\textit{iff}\quad & \exists o_i \in O \,.\, \forall \tau \in \mathsf{Moore}(O \cup \{o_i'\}, I) \,. \\
& \exists \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,. \\
& \overline{\sigma}(\tau, \mathcal{S}) \models (\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi), && (4.14) \\
\Rightarrow\quad & \exists o_i \in O \,.\, \forall \tau \in \mathsf{Moore}(O, I) \,. \\
& \exists \mathcal{S}' \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,. \\
& \overline{\sigma}(\tau, \mathcal{S}) \models (\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi), && (4.15)
\end{aligned}
$$

which contradicts Equation 4.2. (4.9)⇔(4.10) holds because of Lemma 5. and (4.11)⇔(4.12) holds because $\delta \wedge \varphi$ does not contain o_i', so $\mathcal{S}'$ can be $\mathcal{S}$ with $o_i' \leftrightarrow o_i$. (4.13)⇔(4.14) holds because of Lemma 5. Finally, (4.14) implies (4.15) because τ has less information in (4.15).

Lemma 10. *Equation 4.7 holds if and only if Equation 4.3 holds.*

Direction ⇒: is obvious because Equation 4.7 is equivalent to Equation 4.2 (Lemma 3) and Equation 4.2 implies Equation 4.3 (see proof for Theorem 4).

Direction ⇐: we show that Equation 4.7 being false *contradicts Equation 4.3 being* true*. Equation 4.7 being* false *implies Equation 4.12 (see above). As $\mathcal{S}' \models (\varphi[o_i \leftarrow o_i'] \wedge \delta \wedge \varphi)$ implies $(\mathcal{S}' \models \varphi[o_i \leftarrow o_i'] \wedge \delta) \wedge \big(\overline{\sigma}(\tau, \mathcal{S}) \models \varphi\big)$ for all $\tau \in \mathsf{Moore}(O \cup \{o_i'\}, I)$ and thus also for all $\tau \in \mathsf{Moore}(O, I)$, Equation 4.3 cannot hold.*

□

In general, the assumption $\mathcal{S}' \models \varphi[o_i \leftarrow o_i']$ is necessary to prevent a faulty system $\mathcal{S}' \not\models \varphi[o_i \leftarrow o_i']$ from compensating the fault $F \models \delta$ such that $\mathcal{S}' \circ F \models \varphi$. For example, given $I = \emptyset$, $O = \{o\}$, $\varphi = \mathsf{G}o$ with $\delta = \mathsf{G}(o \leftrightarrow \neg o')$, Equation 4.1 would be false without $\mathcal{S}' \models \varphi[o_i \leftarrow o_i']$ because there exists an $\mathcal{S}'$ that always sets $o' =$ false, in which case $\mathcal{S}' \circ F$ has o correctly set to true. However, if δ does not reference the hidden signal o', such a fault compensation is not possible.

Thus, for computing our test strategies we will rely on Equation 4.2. To optimize our implementation, we drop the assumption whenever possible.

4.2.2 Fault Model

The description of the fault model δ covers two different aspects of the fault, the kind of fault κ and the frequency of the fault frq, such that $\delta = \mathsf{frq}(\kappa)$. The fault kind κ is an LTL formula that defines which faults we consider. Examples for different fault kinds are (a) $\kappa = \neg o_i$ that describes a stuck-at-0 fault, (b) $\kappa = o_i \leftrightarrow \neg o_i'$ that defines a bit-flip and (c) $\kappa = o_i' \leftrightarrow \mathsf{X}(o_i)$ which describes a delay by one time step. The fault frequency frq on the other hand defines how often this faultkind is expected to be present. Our implementation supports to only provide the κ and use already implemented fault frequencies. In detail, we support the four fault frequencies $\{\mathsf{G}, \mathsf{FG}, \mathsf{GF}, \mathsf{F}\}$:

- Fault frequency G means that the fault is permanent.
- Frequency FG means that the fault occurs from some time step i on permanently. Yet, we do not make any assumptions about the precise value of i.
- Frequency GF states that the fault strikes infinitely often, but not when exactly.
- Frequency F means that the fault occurs at least once.

Among the four fault frequencies we provide is a natural order. A fault κ that occurs permanently (frequency G) is only a special case of the same fault κ occurring from some point onwards (frequency FG), as "some point" in the case of a permanent fault is the first point in time. The fault κ occurring from some point onwards is again a special case of a fault κ occurring infinitely often (frequency GF), because if it appears infinitely often but does not disappear between the occurrences anymore it just occurs from some point onwards. Finally, this is again a special case of κ occurring at least once, because a fault that occurs at least once may also occur infinitely often. Thus, a test strategy that reveals a fault that occurs at least once (without knowing when) will also reveal a fault that occurs infinitely often, a test strategy that can reveal a fault that occurs infinitely often can also reveal a fault occurring from "some point" in time onwards, and so on. We, thus, start our approach with the goal of deriving a test strategy that can reveal faults occurring at the lowest frequency, i.e., faults occurring at least once, and iteratively increase the fault frequency in case we cannot derive a strategy for the previous frequency.

Algorithm 1 SYNTOUTPUTITERATE: Synthesizes adaptive test strategies from an LTL specification for all outputs in O

1: **procedure** SYNTLTLTEST(I, O, φ, κ), **returns**: A set TS of test strategies
2: TS := $\emptyset$
3: **for each** $o_i \in O$ **do**
4: TS := TS $\cup$ SYNTLTLITERATE($I, O, \varphi, o_i, \kappa, \emptyset$);
5: **return** TS

Algorithm 2 SYNTLTLITERATE: Synthesize an adaptive test strategy from an LTL spec by iterating over fault frequencies frq.

1: **procedure** SYNTLTLITERATE($I, O, \varphi, o_i, \kappa, \Theta$), **returns**: A singleton $\{\mathcal{T}\}$ with a test strategy $\mathcal{T}$ on success or $\emptyset$
2: **for each** frq from $(\mathsf{F}, \mathsf{GF}, \mathsf{FG}, \mathsf{G})$ in this order **do**
3: $\mathcal{T} := \mathsf{synt}_p\big(O \cup \{o_i'\}, I, \big(\varphi[o_i \leftarrow o_i'] \wedge \mathsf{frq}(\kappa)\big) \rightarrow \neg\varphi, O, \Theta\big)$
4: **if** $\mathcal{T} \neq \mathsf{unrealizable}$ **then**
5: **return** $\{\mathcal{T}\}$;
6: **return** $\emptyset$

4.2.3 Test Strategy Computation

We build our test strategy computation approach upon Theorem 4, i.e., for targeted outputs we search for a test strategy $\tau_i \in \mathsf{Moore}(O, I)$ such that $\forall \mathcal{S} \in \mathsf{Mealy}(I, O \cup \{o_i'\}) \,.\, \overline{\sigma}(\tau, \mathcal{S}) \models \big((\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi\big)$ holds. We make use of synthesis with partial information to compute a Moore machine $\mathcal{M} \in \mathsf{Moore}(I', O)$ with $I' \subseteq I$ that allows to enforce the desired LTL objective δ in all environments. Remember that a test strategy is a Moore machine with input and output signals swapped. Hence, we try to compute $\tau_i := \mathsf{synt}_p\big(O \cup \{o_i'\}, I, (\varphi[o_i \leftarrow o_i'] \wedge \delta) \rightarrow \neg\varphi, O\big)$ to derive a test strategy. If the computation of τ_i is successful, the test strategy is guaranteed to be universally complete with respect to fault model δ for a system with inputs I, outputs O, and specification φ. In case synt_p returns $\mathsf{unrealizable}$, a test strategy may exist nevertheless, as Theorem 4 only is a sufficient but not a necessary condition. However, if Proposition 7 or Proposition 8 apply, the method is both sound and complete and, thus, if the algorithm returns $\mathsf{unrealizable}$ there also exists no strategy.

Algorithm. In Algorithm 1, which uses Algorithm 2, we formalize our approach. Let I be the inputs and O be the outputs of the SUT. Moreover, let φ be the specification in LTL of the SUT and let κ be an LTL formula that

describes the kind of fault. Then the result of this algorithm is a test suite TS. To compute TS, the algorithm iterates over all outputs $o_i \in O$ (Line 3 of Algorithm 1) and Algorithm 2 iterates over our four fault frequencies (Line 2), starting with the lowest one, i.e., a fault that occurs at least once. Line 3 attempts to derive a strategy that is capable of revealing every fault that satisfies the provided fault kind κ for the current fault frequency. If the computation is successful, the strategy is added to TS and the next output is processed. Otherwise, the fault frequency is increased and the algorithm again tries to compute a strategy.

Sanity checks. For an unrealizable specification φ or an unrealizable fault model δ, Equation 4.1 is vacuously satisfied. This would make any strategy be a valid solution. To avoid getting such spurious results, we perform a sanity check and test if specification φ and the fault model $\mathsf{G}(\kappa)$ are (Mealy) realizable. To test $\mathsf{G}(\kappa)$ for realizability is enough, because if $\mathsf{G}(\kappa)$ is realizable then so are $\mathsf{FG}(\kappa)$, $\mathsf{GF}(\kappa)$ and $\mathsf{F}(\kappa)$.

Handling unrealizability. Whenever our algorithm returns unrealizable on Line 3 of Algorithm 2 for $\mathsf{frq} = \mathsf{G}$, meaning that we could not even derive a test strategy for a permanent fault, then we print a warning. There are two possible reasons for unrealizability. First, due to limited observability our approach may not be able to compute a strategy although one exists, like in Example 8. And second, there may really be no strategy because there exists an $\mathcal{S}' \mid\!\models \varphi[o_i \leftarrow o_i']$ and $F \mid\!\models \delta$ such that the composition $\mathcal{S} = \mathcal{S}' \circ F$ (see Figure 4.7) is correct, i.e., $\mathcal{S}' \circ F \mid\!\models \varphi$. Less formal, for some realization the fault may not violate the specification, i.e., the fault may behave like an equivalent mutant in mutation testing. For example, consider a stuck-at-0 fault model on output signal o and a correct realization of the specification that never requires this signal to become true. Then such a high degree of underspecification is at least suspicious and may hint to unintended vacuities [17] in specification φ. Hence, it should be investigated manually. If Proposition 7 or 8 applies, then we can be sure that the latter reason applies, i.e. that there exists a high degree of underspecification. The user then may compute some diagnostic information [71] to help him or her understand why no test strategy exists.

Complexity. Both $\mathsf{synt}_p(O, I, \psi, O', \Theta)$ and $\mathsf{synt}(O, I, \psi, \Theta)$ are 2EXPTIME complete in $|\psi|$ [72], so the execution time of Algorithm 2, and consequently also Algorithm 1, are at most doubly exponential in $|\varphi| + |\kappa|$.

Theorem 11. *For a system with inputs I, outputs O, and LTL specification φ over $I \cup O$, if the fault kind κ is of the form $\kappa = \psi$ or $\kappa = (o_i' \leftrightarrow \psi)$, where ψ is an LTL formula over I and O,* SyntLtlTest(I, O, φ, κ) *will return*

a universally complete test suite with respect to the fault model $\delta = \mathsf{G}(\kappa)$ *if such a test suite exists.*

Proof. Since $\mathsf{G}(\kappa)$ implies $\mathsf{frq}(\kappa)$ for all $\mathsf{frq} \in \{\mathsf{F}, \mathsf{GF}, \mathsf{FG}, \mathsf{G}\}$, Theorem 4 and the guarantees of synt_p entail that the resulting test suite TS is universally complete with respect to $\delta = \mathsf{G}(\kappa)$ if $|\mathrm{TS}| = |O|$, i.e., if SyntLtlTest found a strategy for every output. It remains to be shown that $|\mathrm{TS}| = |O|$ for $\kappa = \psi$ or $\kappa = (o_i' \leftrightarrow \psi)$ if a universally complete test suite for $\delta = \mathsf{G}(\kappa)$ exists: either Proposition 7 or Proposition 8 states that Equation 4.2 holds with $\delta = \mathsf{G}(\kappa)$. Thus, synt_p cannot return unrealizable in SyntLtlIterate with $\mathsf{frq} = \mathsf{G}$, so $|\mathrm{TS}|$ must be equal to $|O|$ in this case. □

Theorem 11 states that SyntLtlTest is not only sound but also complete for many interesting fault models such as stuck-at faults or permanent bit-flips. For $\kappa = \psi$, Theorem 11 can even be strengthened to hold for all $\delta = \mathsf{frq}(\kappa)$ with $\mathsf{frq} \in \{\mathsf{F}, \mathsf{GF}, \mathsf{FG}, \mathsf{G}\}$.

4.2.4 Extensions and Variants

A successful test strategy computation results in a universally complete test suite that can detect the specified fault. As a tester we may not only be interested in this specific fault but in any fault that results in the same failure as the specified fault. Such a fault may only get triggered on a specific path through the system. Thus, identifying paths that enforce a failure such as specified in the fault model is also of interest. We achieve this by generalizing the computed strategy, i.e., we remove assignments that are not needed to enforce the desired behavior on the system, and then computing aother strategy for the given fault model that is different from previously derived (generalized) strategies.

User-specified fault frequencies. The user can not only choose from provided fault models (stuck-at-0, stuck-at-1, bitflip, timeshift) and fault frequencies (G, FG, GF, and F), but is free to provide an LTL file specifying the intended faulty behavior. He or she can either specify a kind of fault κ and iterate over the available fault frequencies or choose to treat the provided specification as $\delta = \mathsf{frq}(\kappa)$. The latter option allows the user to also specify other fault frequencies that differ from the provided frequencies, he or she may for example target specific time steps.

Algorithm 2 supports full LTL and thus we can extend the procedure by replacing Line 2 by "**for each** frq from frq in this order", where frq is an additional parameter provided by the user.

Multiple faults and faults at the inputs. While we have presented the approach for a single fault on the output so far, our approach can also handle simultaneous faults at multiple inputs and/or outputs. For example, consider simultaneous faults on the outputs $\{o_1, \ldots, o_k\} \subseteq O$ with every faulty output being described in its own fault model such that the final fault model becomes $\delta = \bigwedge_{i=1}^{k} \delta_i$. To compute a test strategy that is capable of revealing all faults simultaneously, the synthesis procedure is called as follows:

$$\tau := \mathsf{synt}_p\big(O \cup \{o'_1, \ldots, o'_k\}, I, (\varphi[o_1 \leftarrow o'_1, \ldots, o_k \leftarrow o'_k] \wedge \bigwedge_{i=1}^{k} \delta_i) \rightarrow \neg\varphi, O\big).$$

When considering faults at the inputs, we have to modifiy Line 3 in Algorithm 1 to "**for** each $o \in I \cup O$ **do**" such that we not only compute strategies for output signals but also for input signals. Remember, however, that for a fault model that considers one or more input signals only, the resulting strategy will be the realization of the negated fault model if possible. In the next paragraph we present an enhancement that allows the user to provide an arbitrary fault model in LTL and derive strategies that reveal the specified faults.

Faults within the SUT. If an implementation does not satisfy the fault assumption, the enforced execution by the test strategy may nevertheless reveal (unknown) faults that result in the same failure as the given fault model (see Example 9).

Example 9. Consider a system with input $I = \{i\}$, output $O = \{o\}$, and specification $\varphi = \mathsf{G}((i \leftrightarrow \mathsf{X}\neg i) \rightarrow \mathsf{X}o)$. The specification enforces o to be set to true whenever input i alternates between true and false in consecutive time steps. Consider a stuck-at-0 fault $\delta = \mathsf{GF}\neg o$ at the output o. The test suite TS $= \{\mathcal{T}_6\}$ with the test strategy $\mathcal{T}_6$ illustrated in Figure 4.9 (on the left) is universally complete with respect to δ. The test strategy $\mathcal{T}_6$ flips input i in every time step and thus forces the system to set $o = \mathsf{true}$ in the second time step. Now consider the concrete and faulty system implementation in Figure 4.9 (on the right) of φ. It does not satisfy the fault assumption we've put on the system, as the faulty parts needs to be entered first. The test strategy $\mathcal{T}_6$, when executed, first follows the bold edge and then remains forever in the same state. As a consequence, the fault in the system implementation, i.e., o stuck-at-0, is not uncovered. Another valid strategy may have flipped the states of strategy $\mathcal{T}_6$, i.e., starting with i set to false in the initial state. This strategy is now capable of uncovering the fault in the implementation.

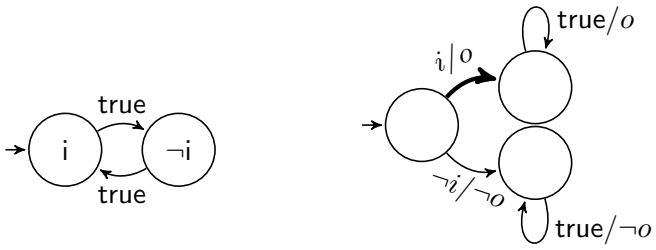


Figure 4.9: Test strategy $\mathcal{T}_6$ and a faulty system implementation of the specification $\varphi = \mathsf{G}((i \leftrightarrow \mathsf{X}(\neg i)) \rightarrow \mathsf{X}(o))$.

Faults within a system implementation can be considered by computing more than one test strategy for a given fault model. We extend Algorithm 1 to generate a bounded number b of test strategies by passing $\Theta = \text{TS}$ in Line 4 and enclosing the line by a **while**-loop that uses an additional integer variable c to count the number of test strategies generated per output o_i. The **while**-loop terminates if no new test strategy is generated or if c becomes equal to b. The resulting test strategies for a certain output and a certain fault model all aim for revealing the same fault, as defined in the fault model. However, every strategy achieves its goal by enforcing a different trace and thus enables the tester to reveal other faults that result in the same failure as the original fault model.

Strategy Generalization. Another optimization we have added to our approach affects the computed test strategy. The synthesis procedure always assigns values to every variable in every state of the strategy and, thus, limits the language of the automaton. However, often it may not be necessary to fix all inputs to force the system into a certain state as shown in Example 10. And as our focus is not only on the specified fault but also on coupled faults, keeping inputs open results in a more general strategy which opens the opportunity to trigger more coupled faults. In other words, we aim for extending the language of the automaton by removing variable assignments when the variable in this state can not generate a counterexample. The tester is then free to assign arbitrary values to the free variables when executing the generalized strategy. This offers the possibility to apply additional testing criteria on the free inputs.

Example 10. Consider an arbiter with inputs $I = \{r_1, r_2\}$, outputs $O = \{g_1, g_2\}$, and specification $\varphi = \big(\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1) \wedge \mathsf{G}(r_2 \rightarrow \mathsf{F}g_2) \wedge \mathsf{G}(\neg g_1 \vee \neg g_2)\big)$. Every request shall eventually be granted and there shall never be two grants at the same time. A valid test strategy τ_7 that tests for a stuck-at-0 fault

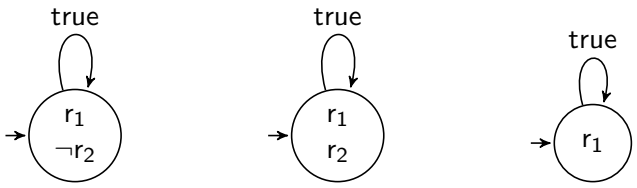


Figure 4.10: Test strategy $\mathcal{T}_7$ on the left, $\mathcal{T}_8$ in the middle and $\mathcal{T}_9$ on the right.

Algorithm 3 GeneralizeStrat: Generalize a strategy.

```
1: procedure Generalize(I, O, φ, o_i, frq, κ, T), returns: A generalization of T
2:     for each q_i ∈ T do
3:         for each x_i ∈ Σ_I do
4:             T' := remove assignment to x_i from state q_i in T
5:             if modelcheckl(T', (φ[o_i ← o_i'] ∧ frq(κ)) → ¬φ) then
6:                 T := T'
7:     return T
```

of signal g_1 from some point in time onwards may simply set $r_1 = \mathsf{true}$ and $r_2 = \mathsf{false}$ all the time (illustrated in Figure 4.10 on the left). This forces the system in every time step to eventually grant this one request. Another valid test strategy τ_8 sets $r_1 = \mathsf{true}$ and $r_2 = \mathsf{true}$ all the time (illustrated in Figure 4.10 in the middle). Now the system has to grant both requests eventually. Both τ_7 and τ_8 test for the defined stuck-at-0 fault of signal g_1 from some point in time onwards but will likely trigger different paths in the SUT. Thus, considering the more general strategy τ_9 that sets $r_1 = \mathsf{true}$ all the time but puts no restrictions on the value of r_2 (illustrated in Figure 4.10 on the right), i.e., the user is free to assign any value to the signal, allows the tester to evaluate different paths in the SUT while still testing for the defined fault class.

Algorithm 3 presents the algorithm to generalize strategy τ. The algorithm loops in Line 2 over all states of τ and in Line 3 over all inputs. In Line 4 the assignment to the input x_i in this state is removed, i.e., made non-deterministic. If the resulting model still satisfies the orginial synthesis formula ψ_{synt}, then τ is overwritten with this new model. Otherwise, the search continues with the previous model.

Note that generalizing a test strategy is a special way of computing multiple concrete test strategies. However, generalization may fail when

computing multiple strategies succeeds, e.g., in Example 9 from the extension for faults within the SUT generalization is not applicable but different strategies can be computed.

If generalization succeeds, the approach provides the user not only with a single strategy for a defined fault class, but with a set of strategies and when computing multiple strategies, we can now immediately exclude the full set.

Optimization for full observability. One optimization we have already discussed at Proposition 8. In Line 3 of Algorithm 2 we can drop part of the assumption and simplify the synthesis step to $\tau_i := \mathsf{synt}\big(O, I, \mathsf{frq}(\kappa) \rightarrow \neg\varphi\big)$ for cases in which κ does not refer to a hidden signal o_i'. Also for a fault model δ that describes a fault of kind $\kappa = (o_i' \leftrightarrow \psi)$, where ψ is an LTL formula over I and O, we can drop the part of the assumption according to Proposition 7 if $\mathsf{frq} = \mathsf{G}$. This simplifies Line 4 of Algorithm 1 to $\tau_i := \mathsf{synt}\big(O, I, \varphi[o_i \leftarrow \psi] \rightarrow \neg\varphi, \Theta\big)$. These simplifications, moreover, no longer require a synthesis procedure with partial information and thus, a larger set of synthesis tools is supported.

Mutating the specification. We can also synthesize adaptive test strategies that would uncover bugs where the SUT implements a mutated (i.e., slightly modified) specification φ' instead of φ by calling $\mathcal{T} := \mathsf{synt}(O, I, \varphi' \rightarrow \neg\varphi, \Theta)$. The implication requires the original specification φ to be violated under the assumption that the mutated specification φ' has been implemented in the SUT. This variant does not require partial information synthesis.

Other specification formalisms. Finally, although we've worked out our approach for LTL, it is not limited to this formal language. The proposed approach works for any other language if (a) the language is closed under Boolean connectives ($\wedge$, $\neg$), (b) the desired fault models can be expressed, and (c) a synthesis procedure (depending on the fault model we may require one supporting partial information) is available.

4.3 Experimental Results

To demonstrate our method we evaluate it on three different specifications, two cases studies that illustrate the applicability on realistic specifications and a smaller toy example that focuses on illustrating the advantages of our method. The first specification is the AMBA Bus Arbiter for two masters [26], the second specification is our toy example of a door system that requires sophisticated strategies to unlock the door, and the third specifica-

Table 4.1: Assumptions of the AMBA Specification.

A1	$\mathsf{G}(\texttt{hbursteqincr} \rightarrow \neg\texttt{hbursteqburst4})$ $\mathsf{G}(\texttt{hbursteqburst4} \rightarrow \neg\texttt{hbursteqincr})$
A2	$\mathsf{G}((\texttt{hmastlock} \wedge \texttt{hbursteqincr} \wedge \neg\texttt{hmaster}) \rightarrow \mathsf{X}(\mathsf{F}(\neg\texttt{hbusreq}_0)))$ $\mathsf{G}((\texttt{hmastlock} \wedge \texttt{hbursteqincr} \wedge \texttt{hmaster}) \rightarrow \mathsf{X}(\mathsf{F}(\neg\texttt{hbusreq}_1)))$
A3	$\mathsf{G}(\mathsf{F}(\texttt{ready}))$
A4	$\mathsf{G}(\texttt{hlock}_0 \rightarrow \texttt{hbusreq}_0)$ $\mathsf{G}(\texttt{hlock}_1 \rightarrow \texttt{hbusreq}_1)$

tion is the FDIR component.

We extended the LTL synthesis tool PARTY [70] and use it for the AMBA case study. PARTY implements SMT-based bounded synthesis [48] for LTL, which sets a bound b on the number of states of the system to be synthesized. This bound is increased iteratively until a solution is found. For the second experiment, where we only use fault models that do not require partial information, we use the synthesis tool Acacia+ [32]. And for the FDIR component, we use again PARTY.

We have split this Section into three parts, first we present the specifications of our two case studies and the toy example, then we discuss the computation of the test strategies for all three examples and, finally, we evaluate the derived test strategies for the AMBA example and the FDIR example on real implementations.

4.3.1 Formal Specifications

In this section we give the LTL specification of the ARM AMBA bus arbiter for two masters and we present the FDIR component of the satellite Eu:CROPIS and formalize its requirements.

AMBA

The ARM AMBA bus arbiter specification is an industrial sized specification formalized in [26] and deriving test cases for such a specification illustrates that our approach can successfully handle real world examples.

The specification for two masters contains 7 input signals $\{\texttt{ready}, \texttt{hlock}_0, \texttt{hlock}_1, \texttt{hbusreq}_0, \texttt{hbusreq}_1, \texttt{hbursteqincr}, \texttt{hbursteqburst4}\}$ and 7 output signals $\{\texttt{hmaster}, \texttt{hgrant}_0, \texttt{hgrant}_1, \texttt{hmastlock}, \texttt{start}, \texttt{locked}, \texttt{decide}\}$. The LTL assumptions are presented in Table 4.1 and the LTL guarantees are presented in Table 4.2.

PIN

This specification is a toy example to illustrate the advantages of our approach. It allows for the system that implements the specification to choose arbitrary binary three digit codes in two consecutive time steps whenever the user wants to unlock the door. To successfully unlock the door, the user has to mirror these codes correctly. The example has 7 input signals {$\texttt{action}_{\text{open}}$, $\texttt{action}_{\text{close}}$, $\texttt{action}_{\text{lock}}$, $\texttt{action}_{\text{unlock}}$, $\texttt{press}_A$, $\texttt{press}_B$, $\texttt{press}_C$} and 5 output signals {`doorclosed`, `doorlocked`, $\texttt{digit}_A$, $\texttt{digit}_B$, $\texttt{digit}_C$}, all of them observable. The assumptions (presented in Table 4.3) assure that the inputs are mutually exclusive and that all inputs eventually reoccur. The guarantees (presented in Table 4.4) of the specification require from any implementation to open and close an unlocked door when requested. The door can be locked by the user and to unlock the door, the user has to successfully mirror in two consecutive time steps a shown code that is freely chosen by the implementation.

FDIR

Often, a satellite contains two redundant control units, such that in case of problems with one unit the satellite can switch to the backup unit. The Fault Detection Isolation and Recovery (FDIR) system is the component of the satellite that monitors the running control unit and decides when to restart the same unit and when to switch to the backup unit. Based on housekeeping signals from the running control unit it may send a message to the electronic power system to switch a unit off or on. When a sever error occurs and no backup unit is available anymore, the FDIR system may require the satellite to be switched to a safe mode from which it can only be recovered by a reset command sent from ground control. A reset message from outside starts one of the two control units and restarts the FDIR system.

In Figure 4.11 an overview of such a composition with an FDIR system is illustrated. While the `FDIR` block illustrates the component itself, `EP` illustrates the electronic power component, S_1 the nominal control unit and S_2 the backup control unit. The FDIR component is connected to the electronic power component and can send messages to switch one of the two control units on or off. It receives housekeeping information from the current running unit. In case of a non critical error, the FDIR system may initiate a restart of the same unit by requesting to switch it off and on again, or it may request to switch to the backup unit by requesting to

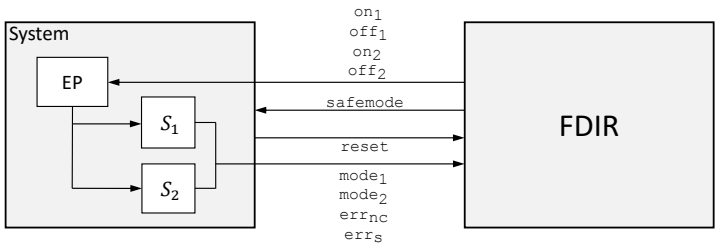


Figure 4.11: An overview of how the FDIR component is integrated in the satellite.

switch the running unit off and the backup unit on. In case of a severe error, a restart of the same unit is not allowed and the FDIR system has to switch off the running control unit and switch to the backup unit. If the FDIR system has already initiated a switch to the backup unit before and there is no available backup unit anymore, it has to activate the safe mode after successfully requesting to switch off the faulty control unit.

Input signals to the FDIR component are $I = \{\texttt{mode}_1, \texttt{mode}_2, \texttt{err}_{\texttt{nC}}, \texttt{err}_{\texttt{crit}}, \texttt{reset}\}$. They are described in Table 4.5. The (observable) output signals of the FDIR component are $O = \{\texttt{on}_1, \texttt{off}_1, \texttt{on}_2, \texttt{off}_2, \texttt{safemode}\}$ and signals that are not observable are $O' = \{\texttt{lastupisnom}, \texttt{allowswitch}\}$. They are described in Table 4.6.

The complete LTL specification of the FDIR component consists of the assumptions A1-A6 and the guarantees G1-G13. All properties are listed in Table 4.7, expressing the following intentions:

A1 Whenever both control units are off, then there is no running unit that can have an error. Thus, the error signals have to be low as well.

A2 The error signals are mutual exclusive. If the environment enforces a reset then both error signals have to be low, because we assume that ground control has taken care of the errors.

A3 After a reset enforced by the environment, one of the two control units has to be running and the other has to be off.

A4 Whenever the FDIR component sends $\texttt{on}_1$, we assume that in the next time step unit number one is running ($\texttt{mode}_1$) and the state of the second unit ($\texttt{mode}_2$) does not change. The same assumption applies analogously for $\texttt{on}_2$.

A5 Whenever the FDIR component sends $\texttt{off}_1$, we assume that in the next time step unit number one is off ($\neg\texttt{mode}_1$) and the state of the second unit ($\texttt{mode}_2$) does not change. The same assumption applies analogously for $\texttt{off}_2$.

A6 We assume that the environment, more specifically the electronic power unit, is not immediately free to change the state of the units when there is no message from the FDIR component. It has to wait for one more time step (with no messages of the FDIR component).

G1 This guarantee keeps track which unit was last activated by the FDIR component.

G2 We require the signals $\texttt{on}_1$, $\texttt{off}_1$, $\texttt{on}_2$ and $\texttt{off}_2$ to be mutually exclusively set to high.

G3 Whenever both units are off, then the FDIR component eventually requests switching on one of the units ($\texttt{on}_1$, $\texttt{on}_2$) or activates `safemode` or observes a `reset`.

G4 We restrict the FDIR component to not enter `safemode` as long as the component can switch to the backup unit.

G5 The FDIR component must not request switching on one of the units ($\texttt{on}_1$, $\texttt{on}_2$) as long as one of the units is running.

G6 Whenever the FDIR component is not allowed to switch to the backup unit, then it must not request switching the backup unit on.

G7 Once the FDIR component switches to the backup unit it is not allowed anymore to switch again (unless the environment performs a reset, see G9).

G8 As long as the FDIR component only restarts the same unit it is still allowed to switch in the future.

G9 A `reset` by the environment allows the FDIR component again to switch to the backup unit if required.

G10 Whenever the FDIR component is in `safemode` it must not request switching on one of the units ($\texttt{on}_1$,$\texttt{on}_2$).

G11 Once a switch is not allowed anymore and the environment does not perform a reset, then the switch is also not allowed in the next time step.

G12 Whenever the FDIR component observes a server error ($\texttt{err}_{\texttt{crit}}$), it must eventually switch to the backup unit or activate `safemode` unless the environment performs a `reset` or the error disappears by itself (without restarting the unit).

G13 Whenever the FDIR component observes a non-critical error ($\texttt{err}_{\texttt{nC}}$), it must eventually switch to the backup unit or activate `safemode` or the error disappears (restarting the currently running unit is allowed).

4.3.2 Test Strategy Generation

AMBA

In 4.3.1 we have given the LTL specification of the ARM AMBA bus arbiter. The properties can be clustered into 3 (interdependent) parts [26]: (a) deciding about the next access, (b) starting an access, and (c) granting the bus. In order to improve scalability and demonstrate that our approach can operate on incomplete specifications, we synthesize test strategies for these 3 parts separately. Each part is combined with all assumptions to ensure that the synthesized test strategies can be run on the entire system.

Table 4.8 summarizes the results for the computation of the strategies. We computed strategies for three different fault models present in the rows. While for the stuck-at-0 and stuck-at-1 fault assumption we have full information we also computed strategies for a bit-flip fault model that requires synthesis with partial information. Sub-rows for every fault model distinguish the output signals $o_i \in O$. The column-blocks contain results for the three specification parts and the full specification. For every computation we present in the sub-columns of the respective specification (a) the lowest fault frequency for which Algorithm 1 found a solution ("-" indicates that no strategy with $\leq b$ states exists, even with $\mathsf{frq} = \mathsf{G}$), (b) the number of states in the resulting test strategy, (c) the execution time, and (d) the peak memory consumption over all outputs. An empty sub-row indicates that the output does not occur in that specification parts.

In many cases, the synthesized test strategies cannot only reveal permanent faults but also transient faults with low frequencies. For stuck-at-0 and stuck-at-1, we can consider the entire spec, and we get such strategies for 12 cases. If we use the fault model that flips the output, we have to restrict ourselves to a subset of the spec. Nevertheless, we can derive another six strategies of the desired quality.

Deriving strategies for parts of the specification that reveal (transient) flips succeeds more often than deriving strategies revealing (transient) stuck-at faults because, with the latter fault model, an output may be (temporarily) stuck at the correct value. On the other hand, synthesizing flip-tests consumes more resources because the optimization discussed in Proposition 8 cannot be applied. This is also the reason for the timeout with flips on the full specification. Although test strategies are found for most outputs when processing the three specification parts separately, processing the full specification yields better strategies but takes longer.

PIN

Although the AMBA specification is industrial, the realization is rather straightforward, which is confirmed by the resulting strategies that all contain at most three states. To evaluate our approach on a specification that requires more complex strategies, we apply it on the PIN toy example that specifies the opening mechanism of a door that is protected by a PIN when the door is locked. We have given the respective LTL specification of it in Section 4.3.1.

Table 4.9 summarizes the results using Acacia+. Our approach is able to derive GF strategies for the output `doorclosed`. For the output `doorlocked`, that is not specified for every step in time, our approach derived strategies detecting FG faults. As the number of states indicates, the strategies are larger, because they need to adapt the input behavior to the observed behavior of the implementation that is free to choose the PIN required for successfully unlocking the door.

FDIR

For the specification of the FDIR system (see Section 4.3.1) we derived for the output signals $\texttt{on}_1$, $\texttt{off}_1$ and `safemode` strategies for the basic fault models stuck-at-0, stuck-at-1 and bit-flip. The signals $\texttt{on}_2$ and $\texttt{off}_2$ are analog to the nominal signals. Thus, the resulting strategies would be equivalent to the strategies for the nominal signals, only with redundant and nominal behavior switched. The results for the strategy generation are presented in Table 4.10.

The more freedom there is for implementations of the specification, the more difficult it becomes to compute a strategy. The search for strategies that are capable of detecting a bit-flip is the most difficult one as we cannot make use of our optimization for full observability of the output signals. For

all signals with a stuck-at-0 fault and for the $\texttt{off}_1$ signal with one of the other two faults we are able to derive test strategies that can detect the fault if it is permanent from some point onwards. For the signals $\texttt{on}_1$ and `safemode` we are able to derive strategies for stuck-at-1 faults and bit-flips also at a lower frequency, i.e., we can detect those faults also if they occur at least infinitely often.

We illustrate and explain one derived strategy in detail. The strategy derived for the signal `safemode` being stuck-at-0 consists of four states. Figure 4.12 illustrates one of the strategies computed with PARTY. In the first state (state 0) we have the nominal control unit running ($\texttt{mode}_1$) and activate the $\texttt{err}_{\texttt{nC}}$ flag, i.e., we raise a non critical error that requires the component to request a restart of the same unit until the error is gone or to request a switch to the backup unit. We do this until the FDIR component requests switching off the nominal unit. In the next state (state 1) we have to wait, doing nothing, for the FDIR component to decide how to proceed. Once the FDIR component requests a switch to the backup unit (state 3), we raise the $\texttt{err}_{\texttt{nC}}$ flag, i.e., we again raise a non critical error, but with the redundant unit running ($\texttt{mode}_2$), until the system requests switching off the redundant unit. It may try to restart the same unit, but we will always raise the error flag until the FDIR component eventually has to activate safe mode if it satisfies the specification.

4.3.3 Evaluation of the Test Strategies

To evaluate the computed test strategies for the AMBA and the FDIR specification, we applied them to real implementations.

AMBA

For the AMBA protocol, we implemented an arbiter for two masters in Verilog. We model checked the implementation against the specification to be sure that the implementation is correct with respect to the specification. Next we generated mutations for every line of code of the correct implementation by fixing assignments in a specified time step to a fixed value to introduce possible transient errors. To eliminate equivalent mutants, we model checked all the mutated implementations and removed those which did not violate the specification. Fixing the assignment to 0 resulted in 39 mutants, fixing the assignment to 1 in 37 and negating the assignment resulted in 41 mutants.

For our test suite TS we used the strategies derived from the full speci-

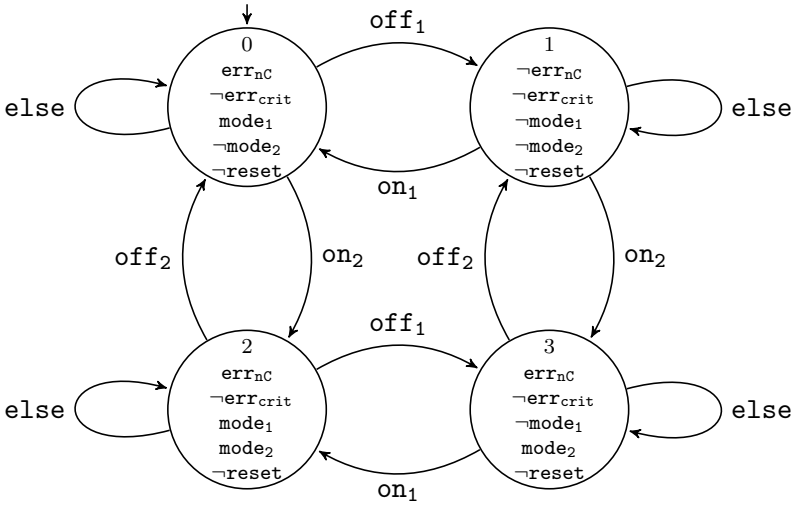


Figure 4.12: Test strategy that forces `satmodesafe` to `true`.

fication (see Table 4.8). The test suite consists of the one strategy that can detect F stuck-at-0 faults, the four strategies that can detect GF stuck-at-0 faults, the two strategies that can detect FG stuck-at-0 faults, the three strategies that can detect F stuck-at-1 faults and the four strategies that can detect GF stuck-at-1 faults. For the test suite gTS we used the generalized strategies from TS. The random test suite consists of nine strategies, i.e., nine different random seeds, that choose valid random input values.

We executed the generalized test strategies from the test suite for a fixed number of time steps and logged the input and output values of the execution traces. Whenever an input signal was not fixed by the strategy we randomly chose a valid value. We then applied the random test suite for the same number of time steps, such that we have a reference to compare to. Every logged trace is then checked against the specification and a mutant is killed whenever the trace is a witness for a violation of the specification, i.e., the specification is violated for any continuation of this trace.

Table 4.11 presents the results. The first two columns show in which time step the mutation is active and which fault was added to the code with *0*, respectively *1*, being an assignment to 0, respectively 1, and with *neg* being a negation of the statement. *Mut* is the number of mutated implementations that violate the specification. *TS#*, respectively *gTS#* and *Rnd#*, present

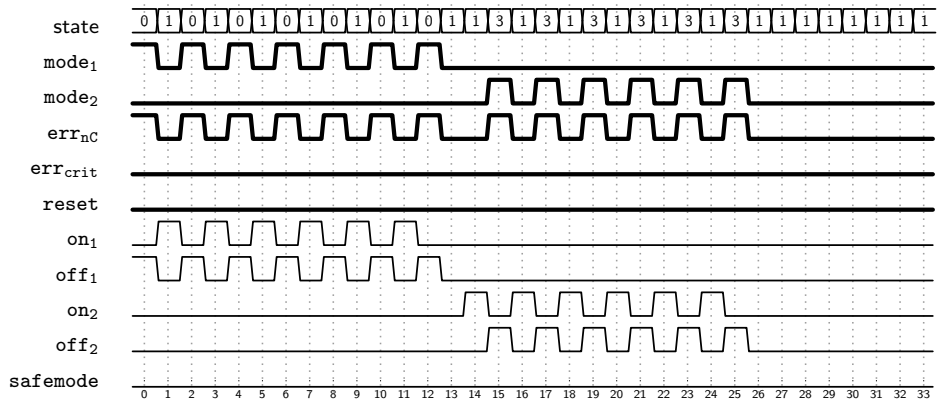


Figure 4.13: Execution trace of a faulty system under the strategy that tests for a stuck-at-0 fault of signal `safemode`. Bold signals are controlled by the strategy.

the number of detected mutants by the strategies of the corresponding test suite with *MS [%]* being the mutation score. For better readability in every row the highest mutation score is highlighted green and the lowest one is highlighted red.

As we have already discussed in Section 4.3.2, the strategies are short and, therefore, the success rate of random strategies for detecting a transient fault comes with no surprise. In the table the advantages of generalization over concrete strategies is immediately obvious. While the computed strategies with concrete values sometimes do have a low detection rate, the generalized strategies detect in all benchmarks at least 40% of the mutants. All detected mutants besides the ones that are part of the fault model for which the strategies were built for are detected due to the coupling effect. For the transient bit-flip we have no strategy at all in our test suite. Nevertheless our generalized strategies detected 41.46% of the mutants in the corresponding benchmark.

FDIR

Test setting. The FDIR component in the Eu:CROPIS satellite is implemented in C++. It is not an exact realization of the specification in Section 4.3.1, because it allows commands to the EP to be lost (e.g. due to electrical faults), which is an extension of the specification.

The implementation uses an abstract interface to access other sub-systems

of the satellite. We replace this interface by a set of test adapters that connect the signals produced by the test strategy. As we are only interested in the functional properties of the implementation, we can run the code on a normal Linux system, instead of the microprocessor which is used in the satellite. This gives access to all Linux based debugging and test tools and allows for example to use `gcov` to measure the code coverage during test execution.

One time step during test execution is split into the following four operations: (1) request values for the input variables I_{FDIR} from the test strategy; (2) provide the values via the test adapter to the FDIR implementation; (3) execute the FDIR implementation for one cycle and (4) extract the output values O_{FDIR} from the test adapter and provide them to the test strategy to derive input values for the next time step. For each time step the execution trace is logged, i.e., we store the values assigned to the inputs I_{FDIR} and outputs O_{FDIR} of the FDIR component.

Mutation testing. We apply mutation analysis to assess the effectiveness, i.e., fault finding abilities, of our test suite. We say a test suite *kills* a mutant program M if it contains at least one test strategy that when executed on M and the original program P produces a trace where at least one output of M differs in at least one time step from the respective output of P (for the same input sequence). A mutant program M is *equivalent* to the original program P if M does not violate the specification. For our evaluation we manually identify and remove equivalent mutants.

We derive mutant programs from the implementation of the FDIR component by systematically introducing the following four mutations in each line: (1) removing the line, (2) replacement of `true` with `false` or `false` with `true`, (3) replacement of `==` with `!=` or `!=` with `==`, and (4) replacement of `&&` with `||` or `||` with `&&`. In total, 198 mutant programs are generated. We use the GNU compiler `gcc` to remove all mutant programs which do not compile and thus not conform to the C++ programming language. Also all mutant programs which fail during runtime e.g. by raising a segmentation fault are removed. We manually analyzed the remaining 96 mutants and identified 23 mutants that are correct with respect to the specification, i.e., equivalent mutants. Thus, 73 mutants violate the specification. Moreover, 11 of these 73 mutants can only violate the specification if the $\texttt{off}_1$ and $\texttt{off}_2$ commands can fail, which contradicts our assumptions on the EP unit. We keep those mutants to check whether the strategies can kill them nevertheless. Next, we execute all test strategies on the mutant programs for 80 time steps each and log the corresponding execution traces.

From the 73 mutants that violate the specification, our strategies all to-

gether are able to kill 52, this corresponds to a mutation score of 71.23%. If we do not take the 11 mutants into consideration that violate our assumptions for the test strategy generation, then the mutation score increases to 80.65%.

We illustrate in Figure 4.13 the execution of the test strategy from Figure 4.12 on a mutant. The strategy aims for revealing a stuck-at-0 fault of signal `safemode`. It can be seen that the test strategy first forces the FDIR component to eventually switch to the backup control unit, the switch happens in time step 14 after several restarts of the control unit. Then the strategy forces the FDIR component to eventually activate `safemode`. However, this mutant is faulty and instead of activating `safemode` the system remains silent from time step 26 onwards. Thus, violating guarantee G3[2].

As the strategies are only derived from requirements, without any implementation specific knowledge, they are applicable on any system that claims to implement the given specification. The mutation score of 71.23% illustrates that our strategies, although computed for only three different faults that are assumed to only affect a single output signal, are also sensitive to many other faults.

If we only apply a single of the four strategies we computed per fault model and output signal, then the resulting test suite can kill (1) 51 mutants, (2) 51 mutants, (3) 49 mutants and (4) 49 mutants. While one strategy per fault and output already achieves a high mutation score, deriving multiple strategies per fault model and output signal still increases the mutation score.

In Table 4.12 we present the mutation score of the individual combinations of signals and fault models. From all the mutants killed, there were 9 mutants only killed by a single signal / fault model combination, namely $\texttt{on}_1$with stuck-at-0 assumption exclusively killing 7 mutants and `safemode`with stuck-at-0 assumption exclusively killing 2 mutants.

Random testing. We compared the fault finding abilities of the generated test strategies and random testing executed for 100, 10'000, and 100'000 time steps, respectively. For random testing we use a similar test setup to the test strategy setup, but instead of requesting the input values I_{FDIR} from a test strategy we use uniformly distributed random values. For each time step, the input and output values are recorded. For each mutant the same input sequence is supplied and the output sequence of the mutant is compared to the output sequence of the actual implementation.

[2]Given that we have decided to we have waited long enough for `safemode` to become true.

Random testing for 100 time steps killed 46 mutants, while random testing for 10'000 time steps killed 69 mutants. Executing the random test for a longer time did not cover any additional mutants, random testing for 100'000 time steps killed 69 mutants as well.

Our strategies are able to kill three mutants that are missed by all of the three random test sequences. These mutants can only be killed when executing certain input/output sequences and it is very unlikely for random testing to hit one of the required sequences. The corresponding sequence requires that a sequence of $\texttt{err}_{\texttt{nC}}$, $\texttt{mode}_1$going low and $\texttt{mode}_1$going high is executed multiple times before either $\texttt{err}_{\texttt{crit}}$or `reset`is triggered.

One mutant is neither covered by the test strategies nor by the random sequences. This mutant requires a longer sequence as well in order to be executed. The mutant is not covered by the test strategies because the sequence is about the timeout of an EP command, which is not covered by the specification from which the test strategies are derived.

Code coverage. Table 4.13 lists the line coverage and branch coverage measured with `gcov` for the different testing approaches. Each line of the table presents one testing approach. The first column contains the name of approach, the second column lists the number of time steps the test was executed, and the third and the fourth column present the achieved line and branch coverage. Overall, the random testing approaches achieve a higher code coverage than the generated adaptive test strategies when executed on the source code of the FDIR component. This stems from the fact that the test strategies are derived from the specification only and independent from a concrete implementation. As mentioned previously, the implementation adds timeouts for operations of the EP, which is not part of the specification. Removing the corresponding instructions would increase the line coverage to 87.3% and the branch coverage to 74.5%. In combination random tests and our strategies together achieve a line coverage of 97.6% and a branch coverage of 87%.

Table 4.2: Guarantees of the AMBA Specification.

G1	$\mathsf{G}(\neg \texttt{ready} \rightarrow \mathsf{X}(\neg \texttt{start}))$
G2	$\mathsf{G}((\texttt{hmastlock} \wedge \texttt{hbursteqincr} \wedge \texttt{start} \wedge \neg\texttt{hmaster}) \rightarrow \mathsf{X}(\neg \texttt{hbusreq}_0 R \neg \texttt{start}))$ $\mathsf{G}((\texttt{hmastlock} \wedge \texttt{hbursteqincr} \wedge \texttt{start} \wedge \texttt{hmaster}) \rightarrow \mathsf{X}(\neg \texttt{hbusreq}_1 R \neg \texttt{start}))$
G3	$\mathsf{G}(\texttt{stateg3}_2 \rightarrow \neg \texttt{start})$
G4,G5	$\mathsf{G}(\texttt{ready} \rightarrow ((\texttt{hgrant}_0 \leftrightarrow \mathsf{X}(\neg \texttt{hmaster})) \wedge$ $(\texttt{hgrant}_1 \leftrightarrow \mathsf{X}(\texttt{hmaster})) \wedge (\texttt{locked} \leftrightarrow \mathsf{X}(\texttt{hmastlock}))))$
G6	$\mathsf{G}((\mathsf{X}\neg \texttt{start}) \rightarrow ((\texttt{hmaster} \leftrightarrow \mathsf{X}(\texttt{hmaster})) \wedge (\texttt{hmastlock} \leftrightarrow \mathsf{X}(\texttt{hmastlock}))))$
G7	$\mathsf{G}((\texttt{decide} \wedge \mathsf{X}(\texttt{hgrant}_0)) \rightarrow (\texttt{hlock}_0 \leftrightarrow \mathsf{X}\texttt{locked}))$ $\mathsf{G}((\texttt{decide} \wedge \mathsf{X}(\texttt{hgrant}_1)) \rightarrow (\texttt{hlock}_1 \leftrightarrow \mathsf{X}\texttt{locked}))$
G8	$\mathsf{G}(\neg \texttt{decide} \rightarrow ((\texttt{hgrant}_0 \leftrightarrow \mathsf{X}(\texttt{hgrant}_0)) \wedge (\texttt{hgrant}_1 \leftrightarrow \mathsf{X}(\texttt{hgrant}_1)) \wedge$ $(\texttt{locked} \leftrightarrow \mathsf{X}(\texttt{locked}))))$
G9	$\mathsf{G}(\texttt{hbusreq}_0 \rightarrow \mathsf{F}(\neg \texttt{hbusreq}_0 \vee \neg \texttt{hmaster}))$ $\mathsf{G}(\texttt{hbusreq}_1 \rightarrow \mathsf{F}(\neg \texttt{hbusreq}_1 \vee \texttt{hmaster}))$
G10	$\mathsf{G}(\neg \texttt{hgrant}_1 \rightarrow (\texttt{hbusreq}_1 R \neg \texttt{hgrant}_1))$ $\mathsf{G}((\texttt{decide} \wedge (\neg \texttt{hbusreq}_0 \wedge \neg \texttt{hbusreq}_1)) \rightarrow \mathsf{X}(\texttt{hgrant}_0))$
CNT	$(\neg \texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0)$ $\mathsf{G}((\neg \texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge$ $\neg(\texttt{hmastlock} \wedge \texttt{hbursteqburst4} \wedge \texttt{start})) \rightarrow \mathsf{X}(\neg \texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0))$ $\mathsf{G}((\neg \texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge \texttt{hmastlock} \wedge$ $\texttt{hbursteqburst4} \wedge \texttt{start} \wedge \neg \texttt{ready}) \rightarrow \mathsf{X}(\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0))$ $\mathsf{G}((\neg \texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge \texttt{hmastlock} \wedge$ $\texttt{hbursteqburst4} \wedge \texttt{start} \wedge \texttt{ready}) \rightarrow \mathsf{X}(\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \texttt{ready}) \rightarrow$ $\mathsf{X}\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \neg \texttt{ready}) \rightarrow$ $\mathsf{X}(\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \texttt{ready}) \rightarrow$ $\mathsf{X}(\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \neg \texttt{ready}) \rightarrow$ $\mathsf{X}(\texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \texttt{ready}) \rightarrow$ $\mathsf{X}(\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \neg \texttt{ready}) \rightarrow$ $\mathsf{X}(\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \texttt{ready}) \rightarrow$ $\mathsf{X}(\neg \texttt{stateg3}_2 \wedge \neg \texttt{stateg3}_1 \wedge \neg \texttt{stateg3}_0))$ $\mathsf{G}((\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \texttt{stateg3}_0 \wedge \neg \texttt{start} \wedge \neg \texttt{ready}) \rightarrow$ $\mathsf{X}(\texttt{stateg3}_2 \wedge \texttt{stateg3}_1 \wedge \texttt{stateg3}_0))$

Table 4.3: Assumptions of the PIN Specification.

A1	$\mathsf{G}(\texttt{action}_{\text{open}} \rightarrow \neg \texttt{action}_{\text{close}} \wedge \neg \texttt{action}_{\text{lock}} \wedge \neg \texttt{action}_{\text{unlock}})$ $\mathsf{G}(\texttt{action}_{\text{close}} \rightarrow \neg \texttt{action}_{\text{open}} \wedge \neg \texttt{action}_{\text{lock}} \wedge \neg \texttt{action}_{\text{unlock}})$ $\mathsf{G}(\texttt{action}_{\text{lock}} \rightarrow \neg \texttt{action}_{\text{close}} \wedge \neg \texttt{action}_{\text{open}} \wedge \neg \texttt{action}_{\text{unlock}})$ $\mathsf{G}(\texttt{action}_{\text{unlock}} \rightarrow \neg \texttt{action}_{\text{close}} \wedge \neg \texttt{action}_{\text{lock}} \wedge \neg \texttt{action}_{\text{open}})$
A2	$\mathsf{GF}\texttt{action}_{\text{open}} \wedge \mathsf{GF}\texttt{action}_{\text{close}} \wedge \mathsf{GF}\texttt{action}_{\text{lock}} \wedge \mathsf{GF}\texttt{action}_{\text{unlock}}$

Table 4.4: Guarantees of the PIN Specification.

G1	$\mathtt{doorclosed} \wedge \mathtt{doorlocked}$
G2	$\mathsf{G}((\mathtt{action}_{\mathrm{open}} \wedge \neg\mathtt{doorlocked}) \rightarrow \mathsf{X}\neg\mathtt{doorclosed})$ $\mathsf{G}((\mathtt{action}_{\mathrm{open}} \wedge \mathtt{doorlocked}) \rightarrow (\mathtt{doorclosed} \leftrightarrow \mathsf{X}\mathtt{doorclosed}))$
G3	$\mathsf{G}(\mathtt{action}_{\mathrm{close}} \rightarrow \mathsf{X}\mathtt{doorclosed})$
G4	$\mathsf{G}(\mathtt{action}_{\mathrm{lock}} \rightarrow \mathsf{X}\mathtt{doorlocked})$
G5	$\mathsf{G}\Big((\mathtt{action}_{\mathrm{unlock}} \wedge$ $(\mathtt{digit}_A \leftrightarrow \mathsf{X}\mathtt{press}_A) \wedge (\mathtt{digit}_B \leftrightarrow \mathsf{X}\mathtt{press}_B) \wedge (\mathtt{digit}_C \leftrightarrow \mathsf{X}\mathtt{press}_C) \wedge$ $(\mathsf{X}\mathtt{digit}_A \leftrightarrow \mathsf{XX}\mathtt{press}_A) \wedge (\mathsf{X}\mathtt{digit}_B \leftrightarrow \mathsf{XX}\mathtt{press}_B) \wedge (\mathsf{X}\mathtt{digit}_C \leftrightarrow \mathsf{XX}\mathtt{press}_C) \wedge$ $\mathsf{XX}\neg\mathtt{action}_{\mathrm{lock}}) \rightarrow \mathsf{XXX}\neg\mathtt{doorlocked}\Big)$ $\mathsf{G}\Big((\mathtt{action}_{\mathrm{unlock}} \wedge$ $\neg((\mathtt{digit}_A \leftrightarrow \mathsf{X}\mathtt{press}_A) \wedge (\mathtt{digit}_B \leftrightarrow \mathsf{X}\mathtt{press}_B) \wedge (\mathtt{digit}_C \leftrightarrow \mathsf{X}\mathtt{press}_C)))$ $\rightarrow \mathsf{XX}\mathtt{doorlocked}\Big)$
G6	$\mathsf{G}((\neg\mathtt{doorclosed} \wedge \neg\mathtt{action}_{\mathrm{close}}) \rightarrow \mathsf{X}\neg\mathtt{doorclosed})$ $\mathsf{G}((\mathtt{doorclosed} \wedge \neg\mathtt{action}_{\mathrm{close}}) \rightarrow \mathsf{X}\mathtt{doorclosed})$

Table 4.5: Descriptions of the input signals of the FDIR component.

$\mathtt{mode}_1$	true iff S_1 is activated
$\mathtt{mode}_2$	true iff S_2 is activated
$\mathtt{err}_{\mathtt{nC}}$	true iff a non-critical error is signaled by S_1 or S_2
$\mathtt{err}_{\mathtt{crit}}$	true iff a severe error is signaled by S_1 or S_2
reset	true iff the FDIR component is reset

Table 4.6: Descriptions of the output signals of the FDIR component.

$\mathtt{on}_1$	true iff S_1 shall be switched on
$\mathtt{off}_1$	true iff S_1 shall be switched off
$\mathtt{on}_2$	true iff S_2 shall be switched on
$\mathtt{off}_2$	true iff S_2 shall be switched off
safemode	true iff the FDIR component initiates the safemode
lastupisnom	true iff the last active unit was S_1 and false if the last active unit was S_2
allowswitch	true iff a switch of S_1 to S_2 or S_2 to S_1 is allowed

Table 4.7: Temporal specification of system-level FDIR component in LTL.

A1	$\mathsf{G}(\neg \mathtt{mode}_2 \wedge \neg \mathtt{mode}_1 \rightarrow \neg \mathtt{err}_{\mathtt{nC}} \wedge \neg \mathtt{err}_{\mathtt{crit}})$
A2	$\mathsf{G}(\neg \mathtt{err}_{\mathtt{nC}} \vee \neg \mathtt{err}_{\mathtt{crit}}) \wedge \mathsf{G}(\mathtt{reset} \rightarrow \neg \mathtt{err}_{\mathtt{nC}} \wedge \neg \mathtt{err}_{\mathtt{crit}})$
A3	$\mathsf{G}(\mathtt{reset} \rightarrow \mathsf{X}(\mathtt{mode}_2 \oplus \mathtt{mode}_1))$
A4	$\mathsf{G}(\neg \mathtt{mode}_1 \wedge \mathtt{on}_1 \wedge \neg \mathtt{off}_1 \wedge \neg \mathtt{on}_2 \wedge \neg \mathtt{off}_2 \wedge \neg \mathtt{reset} \wedge \neg \mathtt{safemode} \rightarrow$ $\mathsf{X}(\mathtt{mode}_1) \wedge (\mathtt{mode}_2 \leftrightarrow \mathsf{X}(\mathtt{mode}_2)))$ $\mathsf{G}(\neg \mathtt{mode}_2 \wedge \neg \mathtt{on}_1 \wedge \neg \mathtt{off}_1 \wedge \mathtt{on}_2 \wedge \neg \mathtt{off}_2 \wedge \neg \mathtt{reset} \wedge \neg \mathtt{safemode} \rightarrow$ $\mathsf{X}(\mathtt{mode}_2) \wedge (\mathtt{mode}_1 \leftrightarrow \mathsf{X}(\mathtt{mode}_1)))$
A5	$\mathsf{G}(\mathtt{mode}_1 \wedge \neg \mathtt{on}_1 \wedge \mathtt{off}_1 \wedge \neg \mathtt{on}_2 \wedge \neg \mathtt{off}_2 \wedge \neg \mathtt{reset} \wedge \neg \mathtt{safemode} \rightarrow$ $\mathsf{X}(\neg \mathtt{mode}_1) \wedge (\mathtt{mode}_2 \leftrightarrow \mathsf{X}(\mathtt{mode}_2)))$ $\mathsf{G}(\mathtt{mode}_2 \wedge \neg \mathtt{on}_1 \wedge \neg \mathtt{off}_1 \wedge \neg \mathtt{on}_2 \wedge \mathtt{off}_2 \wedge \neg \mathtt{reset} \wedge \neg \mathtt{safemode} \rightarrow$ $\mathsf{X}(\neg \mathtt{mode}_2) \wedge (\mathtt{mode}_1 \leftrightarrow \mathsf{X}(\mathtt{mode}_1)))$
A6	$\mathsf{G}((\neg(\neg \mathtt{on}_2 \wedge \neg \mathtt{off}_1 \wedge \neg \mathtt{on}_1 \wedge \neg \mathtt{off}_2) \wedge \mathsf{X}(\neg \mathtt{on}_2 \wedge \neg \mathtt{off}_1 \wedge \neg \mathtt{on}_1 \wedge \neg \mathtt{off}_2) \wedge$ $(\neg \mathtt{reset} \wedge \mathsf{X}(\neg \mathtt{reset}) \wedge \neg \mathtt{safemode} \wedge \mathsf{X}(\neg \mathtt{safemode})) \rightarrow$ $\mathsf{X}((\mathtt{mode}_2 \leftrightarrow \mathsf{X}(\mathtt{mode}_2)) \wedge (\mathtt{mode}_1 \leftrightarrow \mathsf{X}(\mathtt{mode}_1)))$
G1	$\mathsf{G}((\mathtt{on}_1 \wedge \neg \mathtt{on}_2) \rightarrow (\mathsf{X}(\mathtt{lastupisnom})))$ $\mathsf{G}((\neg \mathtt{on}_1 \wedge \mathtt{on}_2) \rightarrow (\mathsf{X}(\neg \mathtt{lastupisnom})))$ $\mathsf{G}((\neg \mathtt{on}_1 \wedge \neg \mathtt{on}_2) \rightarrow (\mathtt{lastupisnom} \leftrightarrow \mathsf{X}(\mathtt{lastupisnom})))$
G2	$\mathsf{G}(\mathtt{on}_1 \rightarrow \neg \mathtt{off}_1 \wedge \neg \mathtt{on}_2 \wedge \neg \mathtt{off}_2)$ $\mathsf{G}(\mathtt{off}_1 \rightarrow \neg \mathtt{on}_1 \wedge \neg \mathtt{on}_2 \wedge \neg \mathtt{off}_2)$ $\mathsf{G}(\mathtt{on}_2 \rightarrow \neg \mathtt{on}_1 \wedge \neg \mathtt{off}_1 \wedge \neg \mathtt{off}_2)$ $\mathsf{G}(\mathtt{off}_2 \rightarrow \neg \mathtt{on}_1 \wedge \neg \mathtt{on}_2 \wedge \neg \mathtt{off}_1)$
G3	$\mathsf{G}(\neg \mathtt{mode}_2 \wedge \neg \mathtt{mode}_1 \rightarrow \mathsf{F}(\mathtt{reset} \vee \mathtt{on}_2 \vee \mathtt{on}_1 \vee \mathtt{safemode}))$
G4	$\mathsf{G}(\mathtt{allowswitch} \rightarrow \neg \mathtt{safemode})$
G5	$\mathsf{G}((\mathtt{mode}_2 \vee \mathtt{mode}_1) \rightarrow \neg \mathtt{on}_1 \wedge \neg \mathtt{on}_2)$
G6	$\mathsf{G}(\neg \mathtt{allowswitch} \wedge \mathtt{lastupisnom} \rightarrow \neg \mathtt{on}_2)$ $\mathsf{G}(\neg \mathtt{allowswitch} \wedge \neg \mathtt{lastupisnom} \rightarrow \neg \mathtt{on}_1)$
G7	$\mathsf{G}(\neg \mathtt{reset} \wedge \mathtt{allowswitch} \wedge \mathtt{lastupisnom} \wedge \mathtt{on}_2 \rightarrow \mathsf{X}(\neg \mathtt{allowswitch}))$ $\mathsf{G}(\neg \mathtt{reset} \wedge \mathtt{allowswitch} \wedge \neg \mathtt{lastupisnom} \wedge \mathtt{on}_1 \rightarrow \mathsf{X}(\neg \mathtt{allowswitch}))$
G8	$\mathsf{G}((\mathtt{allowswitch} \wedge \neg(((\mathtt{lastupisnom} \wedge \mathtt{on}_2) \vee (\neg \mathtt{lastupisnom} \wedge \mathtt{on}_1)))) \rightarrow \mathsf{X}(\mathtt{allowswitch}))$
G9	$\mathsf{G}(\mathtt{reset} \rightarrow \mathsf{X}(\mathtt{allowswitch}))$
G10	$\mathsf{G}(\mathtt{safemode} \rightarrow (\neg \mathtt{on}_1 \wedge \neg \mathtt{on}_2))$
G11	$\mathsf{G}(\neg \mathtt{allowswitch} \wedge \neg \mathtt{reset} \rightarrow \mathsf{X}(\neg \mathtt{allowswitch}))$
G12	$\mathsf{G}((\mathtt{err}_{\mathtt{crit}} \wedge \mathtt{mode}_1 \wedge \neg \mathtt{reset}) \rightarrow \mathsf{F}(\mathtt{reset} \vee \mathtt{safemode} \vee \mathtt{mode}_2 \vee (\mathtt{mode}_1 \mathsf{U}(\mathtt{mode}_1 \wedge \neg \mathtt{err}_{\mathtt{crit}}))))$ $\mathsf{G}((\mathtt{err}_{\mathtt{crit}} \wedge \mathtt{mode}_2 \wedge \neg \mathtt{reset}) \rightarrow \mathsf{F}(\mathtt{reset} \vee \mathtt{safemode} \vee \mathtt{mode}_1 \vee (\mathtt{mode}_2 \mathsf{U}(\mathtt{mode}_2 \wedge \neg \mathtt{err}_{\mathtt{crit}}))))$
G13	$\mathsf{G}((\mathtt{err}_{\mathtt{nC}} \wedge \mathtt{mode}_1 \wedge \neg \mathtt{reset}) \rightarrow \mathsf{F}(\mathtt{reset} \vee \mathtt{safemode} \vee \mathtt{mode}_2 \vee (\mathtt{mode}_1 \wedge \neg \mathtt{err}_{\mathtt{nC}})))$ $\mathsf{G}((\mathtt{err}_{\mathtt{nC}} \wedge \mathtt{mode}_2 \wedge \neg \mathtt{reset}) \rightarrow \mathsf{F}(\mathtt{reset} \vee \mathtt{safemode} \vee \mathtt{mode}_1 \vee (\mathtt{mode}_2 \wedge \neg \mathtt{err}_{\mathtt{nC}})))$

Table 4.8: Results for the AMBA bus arbiter. The suffix "k" multiplies by 10^3.

		Decide Next				Start Access				Grant Bus				Full Spec			
Fault	o_i	frq	$\|\tau\|$	sec	MB	frq	$\|\tau\|$	sec	MB	frq	$\|\tau\|$	sec	MB	frq	$\|\tau\|$	sec	MB
Stuck at 0 ($\kappa = \neg o_i$)	hmaster	FG	2	359	peak: 574 MB	-	-	147	peak: 138 MB	-	-	146	peak: 131 MB	GF	2	4,848	peak: 2,207 MB
	$\texttt{hgrant}_0$	F	2	18						G	2	150		F	2	2,082	
	$\texttt{hgrant}_1$	-	-	856						-	-	172		GF	2	4,991	
	hmastlock	-	-	803		-	-	133		-	-	133		GF	2	5,808	
	start					G	2	126		G	2	230		FG	2	9,367	
	locked	-	-	736						-	-	170		GF	2	5,236	
	decide	G	2	689										FG	2	9,934	
Stuck at 1 ($\kappa = o_i$)	hmaster	FG	2	1,237	peak: 783 MB	G	2	133	peak: 130 MB	G	2	153	peak: 131 MB	F	2	2,388	peak: 1,917 MB
	$\texttt{hgrant}_0$	-	-	6,775						-	-	171		GF	2	5,681	
	$\texttt{hgrant}_1$	F	2	19						G	2	151		F	2	1,970	
	hmastlock	G	2	9,64		G	2	115		G	2	186		F	2	1,473	
	start					GF	3	53		-	-	129		GF	2	5,934	
	locked	GF	2	800						-	-	202		GF	2	5,423	
	decide	-	-	1,011										GF	2	4,169	
Flip ($\kappa = o_i \leftrightarrow \neg o_i'$)	hmaster	G	2	22k	peak: 6,176 MB	G	2	54k	peak: 472 MB	GF	2	1,828	peak: 1,476 MB	Timeout ($>$ 6 days for first output)			
	$\texttt{hgrant}_0$	F	2	29						F	2	10					
	$\texttt{hgrant}_1$	F	2	38						F	2	10					
	hmastlock	G	2	3,385		G	2	53k		GF	2	1,057					
	start					FG	2	43k		G	2	163					
	locked	GF	2	1,525						GF	2	86					
	decide	F	3	61													

Table 4.9: Results for the door specification.

Fault	o_i	frq	$\|\tau\|$	sec	MB
stuck-at-0					
	doorclosed	GF	25	22,341	347
	doorlocked	FG	29	2,425	285
stuck-at-1					
	doorclosed	GF	45	23,290	1,000
	doorlocked	FG	52	3.100	148

Table 4.10: Results for the FDIR specification with max. 4 strategies. The suffix "k" multiplies by 10^3.

Fault	o_i	frq	$\|\tau\|$	sec	peak MB
Stuck 0	$\texttt{on}_1$	FG	4	1.2k	400
	$\texttt{off}_1$	FG	3	517	396
	safemode	FG	4	934	324
Stuck 1	$\texttt{on}_1$	GF	4	438	222
	$\texttt{off}_1$	FG	4	753	378
	safemode	GF	3	169	192
Flip	$\texttt{on}_1$	GF	4	26k	3.6k
	$\texttt{off}_1$	FG	4	98.9k	4.3k
	safemode	GF	3	13.1k	4.3k

Table 4.11: Testing mutated AMBA implementations.

Tick	Fault	Mut	TS#	MS[%]	gTS#	MS[%]	Rnd#	MS[%]
7	0	39	11	28.21	16	41.03	14	35.90
5	1	37	15	40.54	15	40.54	9	24.32
12	1	37	13	35.14	15	40.54	12	32.43
12	neg	41	13	31.71	17	41.46	18	43.90
4, 13	0	39	17	43.59	17	43.59	16	41.03
6, 11	1	37	18	48.65	18	48.65	12	32.43

Table 4.12: Mutation coverage by fault models and signals when executing all four derived strategies.

Output	**Fault Model**			
	S-a-0 [%]	**S-a-1** [%]	**Bit-Flip** [%]	**All** [%]
$\texttt{on}_1$	65.75	39.73	5.48	65.75
$\texttt{off}_1$	5.48	4.11	9.59	9.59
safemode	61.64	6.85	6.85	61.64
All	71.23	39.73	9.59	71.23

Table 4.13: Code coverage.

Approach	**Time steps**	**Line coverage** [%]	**Branch coverage** [%]
Random	100	80.5	64.8
Random	10'000	96.3	85.2
Random	100'000	96.3	85.2
Test strategy	80	76.8	64.8
Together		97.6	87.0

Chapter 5

Finite LTL Interpretation

> "Learning how to think" really means learning how to exercise some control over how and what you think. It means being conscious and aware enough to choose what you pay attention to and to choose how you construct meaning from experience. Because if you cannot or will not exercise this kind of choice in adult life, you will be totally hosed.
>
> David Foster Wallace

This chapter presents the newest work [13] that is yet under review. References to this paper are not made explicit.

When testing a system that implements a specification given in Linear Temporal Logic (LTL), it is not always straightforward to evaluate the resulting traces of the System Under Test (SUT). While LTL is specified on infinite paths, any trace of an executed test is finite. We may need to draw a verdict on whether the system satisfies or violates the property "p holds infinitely often." The problem is that there always exists a continuation of a finite trace that satisfies the property and a different continuation that violates it.

Thus, we present in this chapter a method to evaluate inconclusive finite traces whether they (presumably) satisfy given LTL properties or not. Our approach decides based on observed behavior that is hidden in the trace

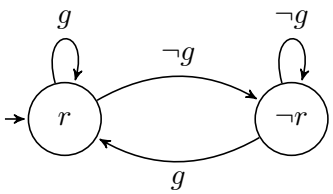


Figure 5.1: Strategy that tests for a stuck-at-0 fault in any system that implements the property $\mathsf{G}\Big(r \rightarrow \mathsf{F}g\Big)$.

whether a trace that is inconclusive, i.e., a trace that has neither satisfied nor violated the specification yet, will presumably satisfy/violate the specification. In Section 5.1 we first motivate our approach as a requirement for the proposed test strategy computation approach from the previous chapter. We then present in Section 5.2 our new approach and evaluate the approach on examples in Section 5.3.

5.1 Motivation

Assume a specification that requires a system to eventually provide a grant g whenever a user triggers a request r. The formalization of this property in LTL looks as follows:

$$\psi = \mathsf{G}\Big(r \rightarrow \mathsf{F}g\Big).$$

Now consider we have used our approach from Chapter 4 and computed a strategy that aims to reveal a fault of the type $\mathsf{FG}\neg g$, i.e., we test whether there eventually exists a persistent stuck-at-0 fault at signal g. Executing the computed strategy (illustrated in Figure 5.1) on two different systems, we present in Table 5.1 the observed trace π_1 of the first system and the observed trace π_2 of the second system.

To evaluate the traces we use existing approaches that evaluate LTL properties on finite traces as discussed in Section 1.3. Again, all the discussed methods evaluate both the traces to the same verdict. However, this is not what we would expect, as the system that produces trace π_1 looks totally fine, whereas something seems to be wrong with the system that produces trace π_2. While requests seem to be granted after exactly two time steps, this is not the case for the third request. This request is not granted

Table 5.1: Observed traces π_1 and π_2 for $\mathsf{G}\Big(r \rightarrow \mathsf{F}g\Big)$.

trace	t	1	2	3	4	5	6	7	8	9	10	11	12	13
π_1	r	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$
	g	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$
π_2	r	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	$-$	$-$	$-$	$-$	$-$
	g	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	$-$	$-$	$-$	$-$	$-$	$-$

in any of the six time steps after the last request, which is much longer than the time it took in the past to observe the grant. Thus, we desire a semantics that evaluates LTL properties on finite traces with respect to observed past behavior. If the time is just too short to observe the satisfaction of the property with respect to previous satisfactions, then we assume that a continuation on the same system would satisfy the property, such as in trace π_1, where we would expect to observe a grant if we continue the trace for two more time steps. If, however, the property is not satisfied for a longer time than the longest witness for a satisfaction in the past (such as in trace π_2), then we conclude this to be bad.

5.2 Counting Semantics for LTL

Before we introduce our counting semantics for LTL in Section 5.2.2, we provide necessary definitions in Section 5.2.1. Then we present in Section 5.2.3 our evaluation method that maps the counting semantics to a truth value.

5.2.1 Definitions

We extend the set of natural numbers (incl. 0) with the two special symbols ∞ (infinite) and $-$ (impossible) and refer to it as $\mathbb{N}_+ = \mathbb{N}_0 \cup \{\infty, -\}$. We define an order on it such that for all $n \in \mathbb{N}_0$, we have $n < \infty < -$. To add two elements $a, b \in \mathbb{N}_+$ we define the addition-operator $\oplus$ as follows:

Definition 12 (Operator $\oplus$)**.** *We define the binary operator* $\oplus : \mathbb{N}_+ \times \mathbb{N}_+ \rightarrow \mathbb{N}_+$ *such that for every* $a, b \in \mathbb{N}_+$*:*

$$a \oplus b \quad = \quad \begin{cases} a + b & \text{if } a, b \in \mathbb{N}_0 \\ \max\{a, b\} & \text{otherwise} \end{cases}$$

To express our counting finite semantics we use pairs (s, f) with $s, f \in \mathbb{N}_+$ and define the following operations on the pairs:

Definition 13 (Operations $\sim$, $\oplus$, $\sqcup$, $\sqcap$)**.** *Given two pairs $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$ and $(s', f') \in \mathbb{N}_+ \times \mathbb{N}_+$ and let $k \in \mathbb{N}_0$, we have:*

$$\sim(s, f) = (f, s) \tag{5.1}$$
$$(s, f) \oplus k = (s \oplus k, f \oplus k) \tag{5.2}$$
$$(s, f) \sqcup (s', f') = (\min(s, s'), \max(f, f')) \tag{5.3}$$
$$(s, f) \sqcap (s', f') = (\max(s, s'), \min(f, f')) \tag{5.4}$$

In Equation 5.1 (operator $\sim$) we define the *swap* between the two values of a pair. The operator $\oplus 1$ in Equation 5.2 defines the increment of both values in the pair by the value 1. The binary operator $\sqcup$ (we refer to it also as *minmax*) in Equation 5.3, computes a new pair with the minimum of the first values as the new first value and the maximum of the second values as the new second value. The binary operator $\sqcap$ (we refer to it also as *maxmin*) in Equation 5.4 is symmetric to the *minmax* operator, i.e., it computes the maximum of the first values as the new first value and the minimum of the second values as the new second value. We now give some examples to illustrate some operations on pairs:

Example 11. Given the pairs $(0, 0)$, $(\infty, 1)$ and $(7, -)$ we have:

$$\begin{array}{rclrcl} \sim(0,0) & = & (0,0) & \sim(\infty,1) & = & (1,\infty) \\ (0,0) \oplus 1 & = & (1,1) & (\infty,1) \oplus 1 & = & (\infty,2) \\ (0,0) \sqcup (\infty,1) & = & (0,1) & (\infty,1) \sqcup (7,-) & = & (7,-) \\ (0,0) \sqcap (\infty,1) & = & (\infty,0) & (\infty,1) \sqcap (7,-) & = & (\infty,1) \end{array}$$

Remark. Note that $\mathbb{N}_+ \times \mathbb{N}_+$ forms a lattice where $(s, f) \trianglelefteq (s', f')$ when $s \geq s'$ and $f \leq f'$ with join $\sqcup$ and meet $\sqcap$. Intuitively, larger values are closer to true.

5.2.2 Counting Semantics

With operations on the pairs defined, we now introduce our counting semantics for LTL. For an arbitrary position i of a given finite trace $\pi \in \Sigma^*$ and a given LTL formula ϕ we give a pair $(s, f) \in \mathbb{N}_+ \times \mathbb{N}_+$. We refer to s as *satisfaction witness count* and to f as *violation witness count*. Intuitively, the value s (f) denotes the number of additional steps needed to witness the satisfaction (violation) of the formula. The value ∞ is used to denote that

the property cannot be satisfied (violated) by a finite continuation and $-$ denotes that the property cannot be satisfied (violated) by any continuation of the trace.

Definition 14 (Counting finitary semantics)**.** *Let $\pi \in \Sigma^*$ be a finite trace, $i \in \mathbb{N}_{>0}$ be the ith position of a trace and $\phi \in \Phi$ be an LTL formula. We define the counting finitary semantics of LTL as the function $d_\pi : \Phi \times \mathbb{N}_{>0} \to \mathcal{P}(\mathbb{N}_+ \times \mathbb{N}_+)$ such that:*

$$
\begin{aligned}
d_\pi(p, i) &= \begin{cases} (0, -) & \text{if } i \leq |\pi| \wedge p \in \pi_i \\ (-, 0) & \text{if } i \leq |\pi| \wedge p \notin \pi_i \\ (0, 0) & \text{if } i > |\pi|, \end{cases} \\
d_\pi(\neg\phi, i) &= \sim d_\pi(\phi, i), \\
d_\pi(\phi_1 \vee \phi_2, i) &= d_\pi(\phi_1, i) \sqcup d_\pi(\phi_2, i), \\
d_\pi(\mathsf{X}\phi, i) &= d_\pi(\phi, i+1) \oplus 1, \\
d_\pi(\phi \mathsf{U} \psi, i) &= \begin{cases} d_\pi(\psi, i) \sqcup \Big(d_\pi(\phi, i) \sqcap d_\pi(\mathsf{X}(\phi \mathsf{U} \psi), i) \Big) & \text{if } i \leq |\pi| \\ d_\pi(\psi, i) \sqcup \Big(d_\pi(\phi, i) \sqcap (-, \infty) \Big) & \text{if } i > |\pi|, \end{cases} \\
d_\pi(\mathsf{F}\phi, i) &= \begin{cases} d_\pi(\phi, i) \sqcup d_\pi(\mathsf{X}\mathsf{F}\phi, i) & \text{if } i \leq |\pi| \\ d_\pi(\phi, i) \sqcup (-, \infty) & \text{if } i > |\pi|. \end{cases}
\end{aligned}
$$

Proposition The evaluation of a proposition for a position inside the trace is trivial, as the proposition either holds or not. If the proposition holds, then we do not need any additional steps to observe satisfaction, i.e., s is 0, and it is impossible to violate it, i.e., f is $-$. In case the proposition does not hold, we have the symmetric witness counts. For the evaluation of the empty word, we take an optimistic view and assume that we can either satisfy or violate the proposition right away, i.e., with 0 additional steps.

Negation Negating a formula simply swaps the witness counts. If we witness the satisfaction of ϕ in n steps, then we witness the violation of $\neg\phi$ in n steps, and vice versa.

Disjunction For the disjunction we take the shorter satisfaction witness count, because the satisfaction of one subformula is enough to satisfy the property. We take the longer violation witness count, because both subformulas need to be violated to violate the property.

Table 5.2: Request/Acknowledge motivating example with π_1, where EOT indicates the end of the trace, i.e., $i > |\pi|$

	1	2	3	4	5	6	7	EOT
r	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	
g	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	
$d_\pi(r,i)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	(0,0)
$d_\pi(g,i)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	(0,0)
$d_\pi(\neg r,i)$	$(-,0)$	$(0,-)$	$(0,-)$	$(-,0)$	$(0,-)$	$(0,-)$	$(-,0)$	(0,0)
$d_\pi(\mathsf{F}g,i)$	$(2,-)$	$(1,-)$	$(0,-)$	$(2,-)$	$(1,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$d_\pi(r \rightarrow \mathsf{F}g,i)$	$(2,-)$	$(0,-)$	$(0,-)$	$(2,-)$	$(0,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$d_\pi(\mathsf{G}(r \rightarrow \mathsf{F}g),i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)

Next The next operator naturally increases the witness counts by one step.

Eventually We use the rewriting rule $\mathsf{F}\phi \equiv \phi \vee \mathsf{XF}\phi$ to define the semantics of the eventually operator. When evaluating the formula after the end of the trace, we replace the remaining obligation ($\mathsf{XF}\phi$) by ($-$,∞). Thus, $\mathsf{F}\phi$ evaluated on the empty word is satisfied by a suffix that satisfies ϕ, and it is violated only by infinite suffixes.

Until We use the same principle for defining the until semantics that we used for the eventually operator. We use the rewriting rule $\phi\mathsf{U}\psi \equiv \psi \vee (\phi \wedge \mathsf{X}(\phi\mathsf{U}\psi))$. On the empty word, $\phi\mathsf{U}\psi$ is satisfied (in the shortest way) by a suffix that satisfies ψ, and it is violated by a suffix that violates both ϕ and ψ.

To illustrate the use of our counting semantics, we take the motivating example from Table 5.1 and evaluate the trace π_1 with respect to specification ψ in Table 5.2. We see that every proposition evaluates to $(0,-)$ when true. The satisfaction of a proposition that holds at time i is immediately witnessed and it cannot be violated by any suffix. Analogously, a proposition evaluates to $(-,0)$ when false. The evaluations of $\mathsf{F}g$ give the number of steps to the next position in which g holds. For instance, the first time at which g holds is at $i = 3$, hence $\mathsf{F}g$ evaluates to $(2,-)$ at $i = 1$, $(1,-)$ at $i = 2$ and $(0,-)$ at time $i = 3$. We also note that $\mathsf{F}g$ evaluates to $(0,\infty)$ at the end of the trace, because it could be immediately satisfied with the continuation of the trace with g that holds, but could be violated only by an infinite suffix in which g never holds. We finally observe that $\mathsf{G}(r \rightarrow \mathsf{F}g)$ evaluates to (∞,∞) at all positions, because the property can be both satisfied and violated only with infinite suffixes.

The rules in Definition 14 restrict the resulting pairs to certain combinations of values in the pair.

Lemma 15. *Let $\pi \in \Sigma^*$ be a finite trace, ϕ an LTL formula and $i \in \mathbb{N}_{>0}$ an index. We have that $d_\pi(\phi, i)$ is of the following form:*

(s,f)		s: a	b	∞	$-$
f	a				✓
	b		✓	✓	
	∞		✓	✓	
	$-$	✓			

where $a \leq |\pi| - i$ and $b > |\pi| - i$.

Proof. The proof is obtained using structural induction on the LTL formula.

Let $s, f \in \mathbb{N}_0 \cup \{\infty\}$. We first define the following sets:

$$P^+_{i,\pi} = \{\ (s, -) \mid s \leq |\pi| - i\ \},$$
$$P^-_{i,\pi} = \{\ (-, f) \mid f \leq |\pi| - i\ \},$$
$$P^?_{i,\pi} = \{\ (s, f) \mid s, f > |\pi| - i\ \},$$
$$P_{i,\pi} = P^+_{i,\pi} \cup P^-_{i,\pi} \cup P^?_{i,\pi},$$

where $P^+_{i,\pi}$ contains all pairs of the form $(a, -)$, the set $P^-_{i,\pi}$ all pairs of the form $(-, a)$ and the set $P^?_{i,\pi}$ all pairs of the form (b_1, b_2), (b_1, ∞), (∞, b_2) and (∞, ∞), with $a \leq |\pi| - i$ and $b_j > |\pi| - i$ for $j \in \{1, 2\}$.

In order to prove the lemma, we first need the following proposition.

Proposition 16. *Let $\pi \in \Sigma^*$ be a finite trace, ϕ an LTL formula and $i \in \mathbb{N}_{>0}$ an index. Then we have that $\forall i > |\pi|, d_\pi(\phi, i) \in P^?_{i,\pi}$.*

Proof. We use structural induction on the structure of the LTL formula to prove this proposition.

Base case $\varphi ::= p$.
$d_\pi(p, i) = (0, 0) \in P^?_{i,\pi}$ for $i > |\pi|$

Induction step $d_\pi(\varphi, i) \in P^?_{i,\pi} \Rightarrow d_\pi(\neg\varphi, i) \in P^?_{i,\pi}$.
If $d_\pi(\varphi, i) \in P^?_{i,\pi}$ then so is $\sim d_\pi(\varphi, i) \in P^?_{i,\pi}$.

Induction step $d_\pi(\varphi_1, i) \in P^?_{i,\pi}, d_\pi(\varphi_2, i) \in P^?_{i,\pi}, \Rightarrow d_\pi(\varphi_1 \vee \varphi_2, i) \in P^?_{i,\pi}$.
This holds because if $d_\pi(\varphi_1, i) = (s_1, f_1) \in P^?_{i,\pi}$ and $d_\pi(\varphi_2, i) = (s_2, f_2) \in P^?_{i,\pi}$ with $s_1, s_2, f_1, f_2 \in \mathbb{N}_0 \cup \{\infty\}$ and $s_1, s_2, f_1, f_2 > |\pi| - i$, then
$d_\pi(\varphi_1 \vee \varphi_2, i) = (s_1, f_1) \sqcup (s_2, f_2) = (\underbrace{\min(s_1, s_2)}_{> |\pi| - i}, \underbrace{\max(f_1, f_2)}_{> |\pi| - i}) \in P^?_{i,\pi}$.

Induction step $d_\pi(\varphi, i+1) \in P^?_{i+1,\pi} \Rightarrow d_\pi(\mathsf{X}\varphi, i) \in P^?_{i,\pi}$.

If $d_\pi(\varphi, i+1) = (s, f) \in P^?_{i+1,\pi}$ with $s, f \in \mathbb{N}_0 \cup \{\infty\}$ and $s, f > |\pi| - (i+1)$, then we have that

$$d_\pi(\mathsf{X}\varphi, i) = \underbrace{(s, f) \oplus 1}_{s \oplus 1,\, f \oplus 1 > |\pi| - i} \in P^?_{i+1,\pi}.$$

Induction step $d_\pi(\varphi_1, i), d_\pi(\varphi_2, i) \in P^?_{i,\pi}, \Rightarrow d_\pi(\varphi_1 \mathsf{U} \varphi_2, i) \in P^?_{i,\pi}$.

If $d_\pi(\varphi_1, i) = (s_1, f_1) \in P^?_{i,\pi}$ and $d_\pi(\varphi_2, i) = (s_2, f_2) \in P^?_{i,\pi}$ with $s_1, s_2, f_1, f_2 \in \mathbb{N}_0 \cup \{\infty\}$ and $s_1, s_2, f_1, f_2 > |\pi| - i$, then applying the definition of $d_\pi(\phi_1 \mathsf{U} \phi_2, i)$ for $i > |\pi|$ we have that

$$d_\pi(\varphi_1 \mathsf{U} \varphi_2, i) = \underbrace{\Big((s_1, f_1) \sqcup \Big(\underbrace{(s_2, f_2) \sqcap (-, \infty)}_{= (\max(s_2, -), \min(f_2, \infty)) = (-, f_2)} \Big)\Big)}_{= (\min(s_1, -), \max(f_1, f_2)) = (s_1, \max(f_1, f_2)) \in P^?_{i,\pi}} \in P^?_{i,\pi}.$$

Induction step $d_\pi(\varphi, i) \in P^?_{i,\pi} \Rightarrow d_\pi(\mathsf{F}\varphi, i) \in P^?_{i,\pi}$.

If $d_\pi(\varphi, i) = (s, f) \in P^?_{i,\pi}$ with $s, f \in \mathbb{N}_0 \cup \{\infty\}$ and $s, f > |\pi| - i$, then applying the definition of $d_\pi(\mathsf{F}\phi, i)$ for $i > |\pi|$, we have that

$$d_\pi(\mathsf{F}\varphi, i) = d_\pi(\varphi, i) \sqcup (-, \infty) = (\underbrace{\min(s, -)}_{s > |\pi| - i}, \underbrace{\max(f, \infty)}_{\infty > |\pi| - i}) \in P^?_{i,\pi}.$$

□

Now we prove the Lemma by proving the closure of $P_{i,\pi}$ under $d_\pi(\phi, i)$ inductively on the structure of the LTL formula.

Base case $\varphi ::= p \quad d_\pi(p, i) \quad = \quad \begin{cases} (0, -) \in P^+_{i,\pi} & \text{if } i \leq |\pi| \wedge p \in \pi_i \\ (-, 0) \in P^-_{i,\pi} & \text{if } i \leq |\pi| \wedge p \notin \pi_i \\ (0, 0) \in P^?_{i,\pi} & \text{if } i > |\pi| \end{cases}$

Induction step $d_\pi(\varphi, i) \in P_{i,\pi} \Rightarrow d_\pi(\neg\varphi, i) \in P_{i,\pi}$ We have three cases:

$(d_\pi(\varphi, i) \in P^+_{i,\pi})$ $d_\pi(\neg\varphi, i) = \sim(d_\pi(\varphi, i)) \in P^-_{i,\pi}$

$(d_\pi(\varphi, i) \in P^-_{i,\pi})$ $d_\pi(\neg\varphi, i) = \sim(d_\pi(\varphi, i)) \in P^+_{i,\pi}$

$(d_\pi(\varphi, i) \in P^?_{i,\pi})$ $d_\pi(\neg\varphi, i) = \sim(d_\pi(\varphi, i)) \in P^?_{i,\pi}$

Induction step $A = d_\pi(\varphi_1, i) \in P_{i,\pi}, B = d_\pi(\varphi_2, i) \in P_{i,\pi} \Rightarrow d_\pi(\varphi_1 \vee \varphi_2, i) \in P_{i,\pi}$

We have $3^2 = 9$ cases, since A and B can be elements of $P^+_{i,\pi}$, $P^-_{i,\pi}$ or $P^?_{i,\pi}$.

$(A \in P^+_{i,\pi},\ B \in P^+_{i,\pi})$ $A = (s_1, -), B = (s_2, -), \underbrace{A \sqcup B = (\min(s_1, s_2), -)}_{\in P^+_{i,\pi} \subset P_{i,\pi}}$

$(A \in P^-_{i,\pi},\ B \in P^+_{i,\pi})$ $A \sqcup B = B \in P^+_{i,\pi} \subset P_{i,\pi}$

$(A \in P^?_{i,\pi},\ B \in P^+_{i,\pi})$ $A = (s_1, f_1), B = (s_2, -),\ s_1 > |\pi| - i, s_2 \leq |\pi| - i \Rightarrow s_2 < s_1$

$$A \sqcup B = (\min(s_1, s_2), \max(f_1, -)) = (s_2, -) = B \in P^+_{i,\pi} \subset P_{i,\pi}$$

$(A \in P^+_{i,\pi}, B \in P^-_{i,\pi}\ )$ Since $\sqcup$ is commutative see the case $(A \in P^-_{i,\pi},\ B \in P^+_{i,\pi})$

$(A \in P^-_{i,\pi}, B \in P^-_{i,\pi})$ $A = (-, f_1),\ B = (-, f_2), \underbrace{A \sqcup B = \{(-, \max(f_1, f_2))\}}_{\in P^-_{i,\pi} \subset P_{i,\pi}}$

$(A \in P^?_{i,\pi}, B \in P^-_{i,\pi})$ $A = (s_1, f_1), B = (-, f_2),\ f_1 > |\pi| - i, f_2 \leq |\pi| - i \Rightarrow f_1 > f_2$

$$A \sqcup B = (\min(s_1, -), \max(f_1, f_2)) = (s_1, f_1) = A \in P^?_{i,\pi} \subset P_{i,\pi}$$

$(A \in P^+_{i,\pi}, B \in P^?_{i,\pi})$ Since $\sqcup$ is commutative see the case $(A \in P^?_{i,\pi},\ B \in P^+_{i,\pi})$.

$(A \in P^-_{i,\pi}, B \in P^?_{i,\pi})$ Since $\sqcup$ is commutative see the case $(A \in P^?_{i,\pi},\ B \in P^-_{i,\pi})$.

$(A \in P^?_{i+1,\pi}, B \in P^?_{i,\pi})$ $A = (s_1, f_1), B = (s_2, f_2),\ s_1, f_1, s_2, f_2 > |\pi| - i$

$$A \sqcup B = (\min(s_1, s_2), \max(f_1, f_2)) \in P^?_{i,\pi} \subset P_{i,\pi}$$

Induction step $A = d_\pi(\varphi, i+1) \in P_{i+1,\pi} \Rightarrow d_\pi(\mathsf{X}\varphi, i) \in P_{i,\pi}$ We have three cases:

$(A \in P^+_{i+1,\pi})$ $\underbrace{d_\pi(\mathsf{X}\varphi, i) = A \oplus 1 = (s_1 + 1, -)}_{s_1 \leq |\pi| - i - 1 \Rightarrow s_1 + 1 \leq |\pi| - i} \in P^+_{i,\pi}$

$(A \in P^-_{i+1,\pi})$ $\underbrace{d_\pi(\mathsf{X}\varphi, i) = A \oplus 1 = (-, f_1 + 1)}_{f_1 \leq |\pi| - i - 1 \Rightarrow f_1 + 1 \leq |\pi| - i} \in P^-_{i,\pi}$

$(A \in P^?_{i+1,\pi})$ $\underbrace{d_\pi(\mathsf{X}\varphi, i) = A \oplus 1 = (s_1 \oplus 1, f_1 \oplus 1) \in P^?_{i,\pi}}_{s_1 > |\pi| - i - 1 \Rightarrow s_1 \oplus 1 > |\pi| - i, f_1 > |\pi| - i - 1 \Rightarrow f_1 \oplus 1 > |\pi| - i}$

Induction step $A = d_\pi(\varphi, j) \in P_{j,\pi} \Rightarrow d_\pi(\mathsf{F}\varphi, i) \in P_{i,\pi}$**.**

if $i > |\pi| \Rightarrow A \in P^?_{i,\pi}\ \Rightarrow\ d_\pi(\mathsf{F}\varphi, i) \in P^?_{i,\pi} \subset P_{i,\pi}$ (See Prop. 16)

if $i \leq |\pi| \Rightarrow d_\pi(\mathsf{F}\varphi, i) = d_\pi(\phi, i) \sqcup d_\pi(\mathsf{X}(\mathsf{F}\varphi), i)$

$d_\pi(\mathsf{F}\varphi, i+1) \in P_{i+1,\pi} \Rightarrow d_\pi(\phi, i) \sqcup d_\pi(\mathsf{X}(\mathsf{F}\varphi), i) \in P_{i,\pi}$

and we proved for $i+1 > |\pi|$ that $d_\pi(\mathsf{F}\varphi, i+1) \in P^?_{i+1,\pi} \subset P_{i+1,\pi}$.

Induction step $A = d_\pi(\varphi_1, i) \in P_{i,\pi}, B = d_\pi(\varphi_2, i) \in P_{i,\pi} \Rightarrow d_\pi(\varphi_1 \mathsf{U} \varphi_2, i) \in P_{i,\pi}$**.**

$$
\begin{aligned}
i > |\pi| \Rightarrow A, B \in A \in P^?_{i,\pi} \Rightarrow\ & d_\pi(\varphi_1 \mathsf{U} \varphi_2, i) \in P^?_{i,\pi} \subset P_{i,\pi} \text{ (See Prop. 16)} \\
& i \leq |\pi| \Rightarrow d_\pi(\varphi_1 \mathsf{U} \varphi_2, i) = A \sqcup (B \sqcap (d_\pi(\mathsf{X}(\varphi_1 \mathsf{U} \varphi_2), i))) \\
d_\pi(\varphi_1 \mathsf{U} \varphi_2, i+1) \in P_{i+1,\pi} \Rightarrow & A \sqcup (B \sqcap (d_\pi(\mathsf{X}(\varphi_1 \mathsf{U} \varphi_2), i))) \in P_{i,\pi}
\end{aligned}
$$

and we proved for $i+1 > |\pi|$ that $d_\pi(\varphi_1 \mathsf{U} \varphi_2, i+1) \in P^?_{i+1,\pi} \subset P_{i+1,\pi}$.

□

Finally, we relate our counting semantics to the three valued semantics in Lemma 17.

Lemma 17. *Given an LTL formula and a trace $\pi \in \Sigma^*$ where $i \in \mathbb{N}_{>0}$ is an index and ϕ is an LTL formula, we have that*

$$
\begin{aligned}
\mu_\pi(\phi, i) = \top \quad &\leftrightarrow \quad d_\pi(\phi, i) = (a, -), \\
&\quad\ \nexists x < a \,.\, \mu_{\pi'}(\phi, 1) = \top \text{ with } \pi' = \pi_i \cdot \pi_{i+1} \cdot \ldots \pi_{i+x}, \\
\mu_\pi(\phi, i) = \bot \quad &\leftrightarrow \quad d_\pi(\phi, i) = (-, a), \\
&\quad\ \nexists x < a \,.\, \mu_{\pi'}(\phi, 1) = \bot \text{ with } \pi' = \pi_i \cdot \pi_{i+1} \cdot \ldots \pi_{i+x}, \\
\mu_\pi(\phi, i) = ? \quad &\leftrightarrow \quad d_\pi(\phi, i) = (b_1, b_2), \\
&\quad\ \nexists x < b_1 \,.\, \mu_{\pi'}(\phi, 1) = \top \text{ with } \pi' = \pi_i \cdot \pi_{i+1} \cdot \ldots \pi_{i+x}, \\
&\quad\ \nexists y < b_2 \,.\, \mu_{\pi'}(\phi, 1) = \bot \text{ with } \pi' = \pi_i \cdot \pi_{i+1} \cdot \ldots \pi_{i+y},
\end{aligned}
$$

where $a \leq |\pi| - i$ and b_j is either ∞ or $b_j > |\pi| - i$ for $j \in \{1, 2\}$.

Lemma 17 holds because we only introduce the symbol "−" within the trace when a satisfaction (violation) is observed. The proof can be obtained again with structural induction on the LTL formula.

5.2.3 Evaluation

We now present our evaluation function that assigns a truth value to every pair. We use a 5-valued set of truth values consisting of true ($\top$), presumably true ($\top_P$), inconclusive ($?$), presumably false ($\bot_P$) and false ($\bot$) verdicts. We define the following order over these five values:

$$\bot < \bot_P < \ ? < \top_P < \top.$$

We equip this 5-valued domain with the negation ($\neg$) and disjunction ($\vee$) operations, letting $\neg\top = \bot$, $\neg\top_P = \bot_P$, $\neg? = ?$, $\neg\bot_P = \top_P$, $\neg\bot = \top$ and $\phi_1 \vee \phi_2 = \max\{\phi_1, \phi_2\}$. We define other Boolean operators such as conjunction by the usual logical equivalences ($\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$, etc.).

We evaluate a property on a trace to $\top$ ($\bot$) when the satisfaction (violation) can be fully determined from the trace, following the definition of the three-valued semantics μ. Intuitively, this takes care of the case in which the safety (co-safety) part of a formula has been violated (satisfied), at least for properties that are intentionally safe (intentionally co-safe, resp.) [74].

Whenever the truth value is not determined, we distinguish whether $d_\pi(\phi, i)$ indicates the possibility for a satisfaction, respective violation, in finite time or not. For possible satisfactions, respective violations, in finite time we make a prediction on whether past observations support the believe that the trace is going to satisfy or violate the property. If the predictions are not inconclusive and not contradicting, then we evaluate the trace to the (presumable) truth value $\top_P$ or$\bot_P$. If we cannot make a prediction to a truth value, we compute the truth value recursively based on the operator in the formula and the truth values of the subformulas (with temporal operators unrolled).

We use the predicate pred_π to give the prediction based on the observed witnesses for satisfaction. The predicate $\text{pred}_\pi(\phi, i)$ becomes ? when no witness for satisfaction exists in the past. When there exists a witness that requires at least the same amount of additional steps as the trace under evaluation then the predicate evaluates to $\top$. If all the existing witnesses (and at least one exists) are shorter than the current trace, then the predicate evaluates to $\bot$. For a prediction on the violation we make a prediction on the satisfaction of $d_\pi(\neg\phi, i)$, i.e., we compute $\text{pred}_\pi(\neg\phi, i)$.

Definition 18 (Prediction predicate)**.** *Let s, f denote natural numbers and let $s_\pi(\phi, i), f_\pi(\phi, i) \in \mathbb{N}_+$ such that $d_\pi(\phi, i) = \big(s_\pi(\phi, i), f_\pi(\phi, i)\big)$. We define the 3-valued predicate $pred_\pi$ as*

$$pred_\pi(\phi, i) = \begin{cases} \top & \text{if } \exists j < i \,.\, d_\pi(\phi, j) = (s', -) \text{ and } s_\pi(\phi, i) \leq s', \\ ? & \text{if } \nexists j < i \,.\, d_\pi(\phi, j) = (s', -), \\ \bot & \text{if } \exists j < i \,.\, d_\pi(\phi, j) = (s', -) \text{ and }, \\ & \quad s_\pi(\phi, i) > \max_{0 \leq j < i}\{s' \mid d_\pi(\phi, j) = (s', -)\}, \end{cases}$$

For the evaluation we consider a case split among the possible combinations of values in the pairs as presented in Lemma 15.

Definition 19 (Predictive evaluation)**.** *We define the* predictive evaluation *function* $e_\pi(\phi, i)$*, with* $a \leq |\pi| - i$ *and* $b_j > |\pi| - i$ *for* $j \in \{1, 2\}$ *and* $a, b_j \in \mathbb{N}_0$*, for the different cases of* $d_\pi(\phi, i)$*:*

$d_\pi(\phi, i)$		$e_\pi(\phi, i)$
$(a, -)$		$\top$
(b_1, b_2)	*if* $pred_\pi(\phi, i) > pred_\pi(\neg\phi, i)$	$\top_P$
	if $pred_\pi(\phi, i) = pred_\pi(\neg\phi, i)$	$r_\pi(\phi, i)$
	if $pred_\pi(\phi, i) < pred_\pi(\neg\phi, i)$	$\bot_P$
(b_1, ∞)	*if* $pred_\pi(\phi, i) = \top$	$\top_P$
	if $pred_\pi(\phi, i) =?$	$r_\pi(\phi, i)$
	if $pred_\pi(\phi, i) = \bot$	$\bot_P$
(∞, b_1)		$e_\pi(\neg\phi, i)$
(∞, ∞)		$r_\pi(\phi, i)$
$(-, a)$		$\bot$

where $r_\pi(\phi, i)$ *is an auxiliary function defined inductively as follows:*

$$
\begin{aligned}
r_\pi(p, i) &=? \\
r_\pi(\neg\phi, i) &= \neg e_\pi(\phi, i) \\
r_\pi(\phi_1 \vee \phi_2, i) &= e_\pi(\phi_1, i) \vee e_\pi(\phi_2, i) \\
r_\pi(\mathsf{X}^n\phi, i) &= e_\pi(\phi, i + n) \\
r_\pi(\mathsf{F}\phi, i) &= \begin{cases} e_\pi(\phi, i) \vee r_\pi(\mathsf{XF}\phi, i) & \text{if } i \leq |\pi| \\ e_\pi(\phi, i) & \text{if } i > |\pi| \end{cases} \\
r_\pi(\phi_1 \mathsf{U} \phi_2, i) &= \begin{cases} e_\pi(\phi_2, i) \vee (e_\pi(\phi_2, i) \wedge e_\pi(\mathsf{X}(\phi_1 \mathsf{U} \phi_2), i) & \text{if } i \leq |\pi| \\ e_\pi(\phi_2, i) & \text{if } i > |\pi| \end{cases}
\end{aligned}
$$

The predictive evaluation function is symmetric. Hence, $e_\pi(\phi, i) = \neg e_\pi(\neg\phi, i)$ holds.

We refer again to our motivating example from Table 5.1 and evaluate the trace π_1 with respect to the specification ψ. We present the outcome in Table 5.3. Subformula $r \rightarrow \mathsf{F}g$ is predicted to be $\top_P$ at $i = 7$ because there exists a longer witness for satisfaction in the past (e.g., at $i = 1$). Thus, as we do not expected the globally property to be violated, the trace evaluates to $\top_P$, as expected.

Table 5.3: Request/Acknowledge motivating example with π_1.

	1	2	3	4	5	6	7	EOT
r	$\top$	$-$	$-$	$\top$	$-$	$-$	$\top$	
g	$-$	$-$	$\top$	$-$	$-$	$\top$	$-$	
$d_\pi(r,i)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	(0,0)
$e_\pi(r,i)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	?
$d_\pi(g,i)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	(0,0)
$e_\pi(g,i)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	?
$d_\pi(\neg r,i)$	$(-,0)$	$(0,-)$	$(0,-)$	$(-,0)$	$(0,-)$	$(0,-)$	$(-,0)$	(0,0)
$e_\pi(\neg r,i)$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	?
$d_\pi(\mathsf{F}g,i)$	$(2,-)$	$(1,-)$	$(0,-)$	$(2,-)$	$(1,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(\mathsf{F}g,i)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top_P$	$\top_P$
$d_\pi(r \to \mathsf{F}g,i)$	$(2,-)$	$(0,-)$	$(0,-)$	$(2,-)$	$(0,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(r \to \mathsf{F}g,i)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top_P$	$\top_P$
$d_\pi(\mathsf{G}(r \to \mathsf{F}g),i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)
$e_\pi(\mathsf{G}(r \to \mathsf{F}g),i)$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$

In Figure 5.2 we visualize the evaluation of a pair $d_\pi(\phi,i) = (s,f)$ for a fixed ϕ and a fixed position i. On the x-axis is the witness count s for a satisfaction and on the y-axis is the witness count f for a violation. For a value s, respectively f, that is smaller than the length of the suffix starting at position i (with the other value of the pair always being $-$), the evaluation is either $\top$ or $\bot$. Otherwise the evaluation depends on the values s_{max} and f_{max}. These two values represent the largest witness counts for a satisfaction and a violation in the past, i.e., for positions smaller than i in the trace. Based on the prediction function $\text{pred}_\pi(\phi,i)$ the evaluation becomes $\top_P$, ? or $\bot_P$, where ? indicates that the auxiliary function $r_\pi(\phi,i)$ has to be applied. Starting at an arbitrary point in the diagram and moving to the right increases the witness count for a satisfaction while the witness count for a violation remains constant. Thus, moving to the right makes the pair "more false". The same holds when keeping the witness count for a satisfaction constant and moving up in the diagram as this decrease the witness count for a violation. Analogously, moving down and/or left makes the pair "more true" as the witness count for a violation gets larger and/or the witness count for a satisfaction gets smaller.

Our 5-valued predictive evaluation refines the 3-valued LTL semantics.

Theorem 20. *Let ϕ be an LTL formula, $\pi \in \Sigma^*$ and $i \in \mathbb{N}_{>0}$. We have*

$$\begin{aligned}
\mu_\pi(\phi,i) = \ & \top \ \leftrightarrow \ e_\pi(\phi,i) = \ \top, \\
\mu_\pi(\phi,i) = \ & \bot \ \leftrightarrow \ e_\pi(\phi,i) = \ \bot, \\
\mu_\pi(\phi,i) = \ & ? \ \leftrightarrow \ e_\pi(\phi,i) \in \ \{\top_P, \bot_P, ?\}.
\end{aligned}$$

Theorem 20 holds, because the evaluation to $\top$ and $\bot$ is simply the

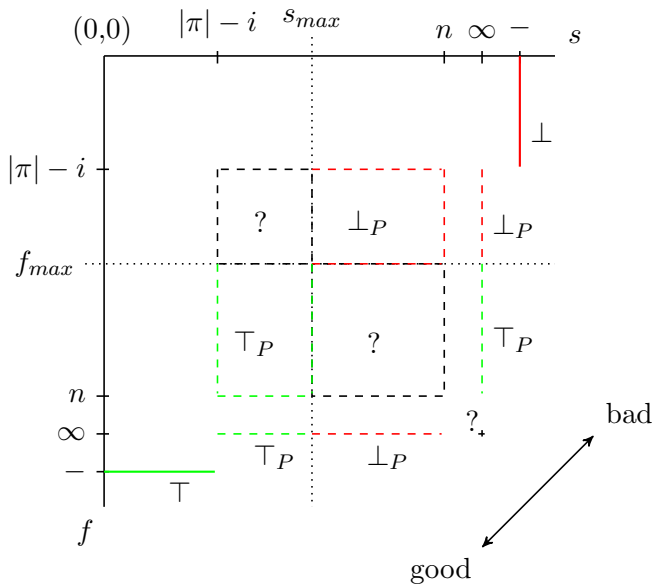


Figure 5.2: Lattice for (s,f) with ϕ and $i < |\pi|$ fixed.

mapping of a pair that contains the symbol "$-$", which we have shown in Lemma 17.

Remember that $\mathbb{N}_+ \times \mathbb{N}_+$ is partially ordered by $\trianglelefteq$. We now show that having a trace that is "more true" than another is correctly reflected in our finitary semantics. To define "more true", we first need the polarity of a proposition in an LTL formula.

Definition 21 (Polarity)**.** *Let $\#\neg$ be the number of negation operators on a specific path in the parse tree of ϕ starting at the root. We define the polarity as the function $pol(p)$ with proposition p in an LTL formula ϕ as follows:*

$$pol(p) = \begin{cases} pos, & \text{if } \#\neg \text{ on all paths to a leaf with proposition } p \text{ is even,} \\ neg, & \text{if } \#\neg \text{ on all paths to a leaf with proposition } p \text{ is odd,} \\ mixed, & \text{otherwise.} \end{cases}$$

With the polarity defined, we now define the constraints for a trace to be "more true" with respect to an LTL formula ϕ.

Definition 22 ($\pi \sqsubseteq_\phi \pi'$)**.** *Given two traces π and π' of equal length and an*

LTL formula ϕ over proposition p, we define that $\pi \sqsubseteq_\phi \pi'$ iff

$$\begin{aligned}\forall i \forall p\,. \quad & pol(p) = mixed \Rightarrow p \in \pi_i \leftrightarrow p \in \pi'_i \text{ and} \\ & pol(p) = pos \Rightarrow p \in \pi_i \rightarrow p \in \pi'_i \text{ and} \\ & pol(p) = neg \Rightarrow p \in \pi_i \leftarrow p \in \pi'_i.\end{aligned}$$

Whenever one trace is "more true" than another, this is correctly reflected in our finitary semantics.

Theorem 23. *For two traces π and π' of equal length and an LTL formula ϕ over proposition p, we have that*

$$\pi \sqsubseteq_\phi \pi' \Rightarrow d_{\pi'}(\phi, 1) \trianglelefteq d_\pi(\phi, 1).$$

Therefore, we have for $\pi \sqsubseteq_\phi \pi'$ that

$$\begin{aligned}& e_\pi(\phi, 1) = \top \Rightarrow e_{\pi'}(\phi, 1) = \top, \text{ and} \\ & e_\pi(\phi, 1) = \bot \Leftarrow e_{\pi'}(\phi, 1) = \bot.\end{aligned}$$

Theorem 23 holds, because we have that replacing an arbitrary observed value in π by one with positive polarity in π' always results with $d_\pi(\phi, 1) = (s, f)$ and $d_{\pi'}(\phi, 1) = (s', f')$ in $s' \leq s$ and $f' \geq f$, as with $\pi \sqsubseteq_\phi \pi'$ we have that π' witnesses a satisfaction of ϕ not later than π and π' also witness a violation of ϕ not earlier than π.

In Table 5.4 we give examples to illustrate the transition of one evaluation to another one. Note that it is possible to change from $\top_P$ to $\bot_P$. However, this is only the predicated truth value that becomes "worse", because we have strengthened the prefix on which the prediction is based on, the values of $d_\pi(\phi, i)$ don't change and remain the same is such a case.

5.3 Examples

Empty Word: The empty word evaluates to ? for all LTL properties. Given that the empty word contains no observation, we do not have any information to predict future events.

Evaluation of the Next Operator: In Table 5.5 and Table 5.6 we illustrate the evaluation of the X operator nested in an F property and nested in a G property.

Our approach focuses on observed past behavior and predicts evaluations of subformulas when possible. The prediction on $\mathsf{X}g$ is necessary to draw a

Table 5.4: Making a system "more true".

ϕ	π	$d_\pi(\phi, 1)$	$e_\pi(\phi, 1)$
p	$-$	$(-, 0)$	$\bot$
	$\top$	$(0, -)$	$\top$
$p \wedge \mathsf{X}\mathsf{F}p$	$- - -$	$(-, 0)$	$\bot$
	$\top - -$	$(3, \infty)$	$\bot_P$
$\mathsf{G}p$	$-\top\top$	$(-, 0)$	$\bot$
	$\top\top\top$	$(\infty, 3)$	$\top_P$
$\mathsf{F}p$	$- - -$	$(3, \infty)$	$\bot_P$
	$\top - -$	$(0, -)$	$\top$
$\mathsf{F}\mathsf{G}p$	$\top - \top - \top$	(∞, ∞)	$\bot_P$
	$\top - \top\top\top$	(∞, ∞)	$\top_P$
$\mathsf{G}\mathsf{F}p$	$- - \top - -$	(∞, ∞)	$\top_P$
	$\top - \top - -$	(∞, ∞)	$\bot_P$
$p \vee \mathsf{X}\mathsf{G}p$	$-\top\top$	$(\infty, 3)$	$\top_P$
	$\top\top\top$	$(0, -)$	$\top$

conclusion on the eventually, respectively globally, property being violated, respectively satisfied. For the trace in Table 5.5 our approach results in the expected presumably false verdict, because we have always observed $\mathsf{X}g$ being violated and we do not expect it to be satisfied. For the trace in Table 5.6 our approach results in the expected presumably true verdict, because we have always observed $\mathsf{X}g$ being satisfied and we do not expect it to be violated.

Request/Acknowledge Properties: As a running example we have already illustrated the evaluation of trace π_1 from the motivation with the property

$$\mathsf{G}(r \rightarrow \mathsf{F}g).$$

We now also evaluate the second trace from the motivation. In Table 5.7 we present the evaluation. While for many positions (like $i = 5$) the signal r dominates (because it is false and, thus, the implication is trivially satisfied) this is not the case for position $i = 4$. At this position the implication is not yet satisfied within the trace and, thus, can be at earliest satisfied in 4 steps by extending the trace with $g = \mathsf{true}$ at $i = 8$. However, the longest

Table 5.5: Evaluation of $\mathsf{FX}g$.

i	1	2	3	4	EOT	
g	$-$	$-$	$-$	$-$		
$d_\pi(g,i)$	$(-,0)$	$(-,0)$	$(-,0)$	$(-,0)$	$(0,0)$	(1)
$e_\pi(g,i)$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot_P$	
$d_\pi(\mathsf{X}g,i)$	$(-,1)$	$(-,1)$	$(-,1)$	$(1,1)$	$(1,1)$	(2)
$e_\pi(\mathsf{X}g,i)$	$\bot$	$\bot$	$\bot$	$\bot_P$	$\bot_P$	
$d_\pi(\mathsf{FX}g,i)$	$(4,\infty)$	$(3,\infty)$	$(2,\infty)$	$(1,\infty)$	$(1,\infty)$	(3)
$e_\pi(\mathsf{FX}g,i)$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	

Table 5.6: Evaluation of $\mathsf{GX}g$.

i	1	2	3	4	EOT	
π_g	$\top$	$\top$	$\top$	$\top$		
$d_\pi(g,i)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,0)$	(1)
$e_\pi(g,i)$	$\top$	$\top$	$\top$	$\top$	$\top_P$	
$d_\pi(\mathsf{X}g,i)$	$(1,-)$	$(1,-)$	$(1,-)$	$(1,1)$	$(1,1)$	(2)
$e_\pi(\mathsf{X}g,i)$	$\top$	$\top$	$\top$	$\top_P$	$\top_P$	
$d_\pi(\mathsf{GX}g,i)$	$(\infty,4)$	$(\infty,3)$	$(\infty,2)$	$(\infty,1)$	$(\infty,1)$	(3)
$e_\pi(\mathsf{GX}g,i)$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	

observed witness for satisfaction of the implication is at $i = 1$ and requires two additional steps. As we've never observed a witness that requires at least 4 additional steps for a satisfaction, the suffix at $i = 4$ is concluded to be presumably false. Hence, the globally property is expected to be violated and we conclude that this trace is going to presumably violate the given property.

Next we illustrate in Table 5.8 why predictions on the different levels of subformulas are necessary. Note that the prediction for the property $\mathsf{F}g$ at Position 5 is $\top_P$, because there exists a witness in the past (at Position 1) that required the same amount of additional steps for satisfaction. when evaluating the property $r \rightarrow \mathsf{F}g$, the prediction for the same Position becomes $\bot_P$, because now the longest witness (at Position 2) only requires one additional step, which is shorter than the required two additional steps (at Position 5). This is, because the signal g is related to the signal r, and at Position 1 the truth value of signal r dominates. Human intuition supports

Table 5.7: Trace π_2 from the motivation.

i	1	2	3	4	5	6	7	EOT	
r	$\top$	$-$	$-$	$\top$	$-$	$-$	$-$		
g	$-$	$-$	$\top$	$-$	$-$	$-$	$-$		
$d_\pi(r,i)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(-,0)$	$(0,0)$	(1)
$e_\pi(r,i)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	?	
$d_\pi(\neg r,i)$	$(-,0)$	$(0,-)$	$(0,-)$	$(-,0)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,0)$	(2)
$e_\pi(\neg r,i)$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	?	
$d_\pi(g,i)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(-,0)$	$(-,0)$	$(0,0)$	(3)
$e_\pi(g,i)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	?	
$d_\pi(\mathsf{F}g,i)$	$(2,-)$	$(1,-)$	$(0,-)$	$(4,\infty)$	$(3,\infty)$	$(2,\infty)$	$(1,\infty)$	$(0,\infty)$	(4)
$e_\pi(\mathsf{F}g,i)$	$\top$	$\top$	$\top$	$\bot_P$	$\bot_P$	$\top_P$	$\top_P$	$\top_P$	
$d_\pi(r \to \mathsf{F}g,i)$	$(2,-)$	$(0,-)$	$(0,-)$	$(4,\infty)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,\infty)$	(5)
$e_\pi(r \to \mathsf{F}g,i)$	$\top$	$\top$	$\top$	$\bot_P$	$\top$	$\top$	$\top$	$\top_P$	
$d_\pi(\mathsf{G}(r \to \mathsf{F}g,i))$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(6)
$e_\pi(\mathsf{G}(r \to \mathsf{F}g,i))$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	

Table 5.8: Need for prediction of individual subformulas.

i	1	2	3	4	5	6	EOT	
r	$-$	$\top$	$-$	$-$	$\top$	$-$		
g	$-$	$-$	$\top$	$-$	$-$	$-$		
$d_\pi(r,i)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(0,0)$	(1)
$e_\pi(r,i)$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	?	
$d_\pi(g,i)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(-,0)$	$(0,0)$	(2)
$e_\pi(g,i)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	?	
$d_\pi(\mathsf{F}g,i)$	$(2,-)$	$(1,-)$	$(0,-)$	$(3,\infty)$	$(2,\infty)$	$(1,\infty)$	$(0,\infty)$	(3)
$e_\pi(\mathsf{F}g,i)$	$\top$	$\top$	$\top$	$\bot_P$	$\top_P$	$\top_P$	$\top_P$	
$d_\pi(r \to \mathsf{F}g,i)$	$(0,-)$	$(1,-)$	$(0,-)$	$(0,-)$	$(2,\infty)$	$(0,-)$	$(0,\infty)$	(4)
$e_\pi(r \to \mathsf{F}g,i)$	$\top$	$\top$	$\top$	$\top$	$\bot_P$	$\top$	$\top_P$	
$d_\pi(\mathsf{G}(r \to \mathsf{F}g),i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(5)
$e_\pi(\mathsf{G}(r \to \mathsf{F}g),i)$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\top_P$	$\top_P$	

this evaluation. While evaluating only $\mathsf{F}g$ allows the observer to conclude that it always takes two additional steps to observe the grant, this is not the case when evaluating $r \to \mathsf{F}g$. For this property, the signal g is only relevant whenever a request r is observed and then the grant g is observed in one additional step.

In another request/acknowledge example we analyze the property

$$\mathsf{G}(r_1 \to \mathsf{F}g_1) \wedge \mathsf{G}(r_2 \to \mathsf{F}g_2)$$

with r_1 being triggered at even time steps, r_2 being triggered at odd time steps, and both requests being always granted after exactly one time step. No matter where you cut the trace there is always one request not yet

Table 5.9: Trace of a system claiming to implement $\mathsf{G}(\neg r_1 \vee \mathsf{F}g_1) \wedge \mathsf{G}(\neg r_2 \vee \mathsf{F}g_2)$.

	1	2	3	4	5	6	7	8	9	10	11	12	13
r_1	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$
g_1	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–
r_2	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–
g_2	–	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$	–	$\top$

granted (Table 5.9 illustrates an example trace).

The two request/grant properties are conjunct on the highest level of the formula. Our approach computes truth values for every subformula, i.e., computes independent predictions for both request/grant properties which is in both cases $\top_P$. On the highest level (no predictions are possible anymore at this level, because all computed pairs are of the form (∞, ∞)) the computed truth values for the two request/grant properties are conjunct and result in the expected verdict presumably true.

Evaluation of the Until Operator: To illustrate our approach on a specification that contains an until operator, we consider the property

$$\mathsf{G}((\mathsf{X}a)\mathsf{U}\mathsf{X}\mathsf{X}b).$$

Table 5.10 shows an example trace and the associated evaluation. The longest observed witness for satisfaction of the until property starts at position 1 and requires six additional time steps. In positions 1, 2, 3 and 4 the subformula $\mathsf{X}a$ holds, until in position 5 the subformula $\mathsf{X}\mathsf{X}b$ holds. The suffix of the trace from position 6 can be satisfied at earliest after 3 time steps by an extension of the trace with $b = \top$ at $i = 9$. As the suffix is shorter than the longest observed witness for satisfaction and we have not observed any violation, this inconclusive suffix is predicted to be presumably true. The same applies for the suffixes starting at $i = 7$ and $i = 8$. Thus, we neither observe nor expect a violation of the globally property. Hence, the property evaluates to $\top_P$ with respect to the given trace.

Stabilization Properties: Consider the property

$$\mathsf{FG}a \vee \mathsf{FG}\neg a$$

that states that eventually the truth value of a has to stabilize.

We analyze the traces presented in Table 5.11. While in trace π_1 the system seems to flip the truth value of a always after time time steps, in

Table 5.10: Evaluation of $\mathsf{G}((\mathsf{X}a)\mathsf{U}\mathsf{X}\mathsf{X}b)$.

i	1	2	3	4	5	6	7	8	EOT	
a	–	$\top$	$\top$	$\top$	$\top$	–	$\top$	$\top$		
b	–	$\top$	–	–	–	–	$\top$	–		
$d_\pi(a,i)$	$(-,0)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(-,0)$	$(0,-)$	$(0,-)$	$(0,0)$	(1)
$e_\pi(a,i)$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	?	
$d_\pi(\mathsf{X}a,i)$	$(1,-)$	$(1,-)$	$(1,-)$	$(1,-)$	$(-,1)$	$(1,-)$	$(1,-)$	$(1,1)$	$(1,1)$	(2)
$e_\pi(\mathsf{X}a,i)$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	?	?	
$d_\pi(b,i)$	$(-,0)$	$(0,-)$	$(-,0)$	$(-,0)$	$(-,0)$	$(-,0)$	$(0,-)$	$(-,0)$	$(0,0)$	(3)
$e_\pi(b,i)$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	?	
$d_\pi(\mathsf{X}b,i)$	$(1,-)$	$(-,1)$	$(-,1)$	$(-,1)$	$(-,1)$	$(1,-)$	$(-,1)$	$(1,1)$	$(1,1)$	(4)
$e_\pi(\mathsf{X}b,i)$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	?	?	
$d_\pi(\mathsf{X}\mathsf{X}b,i)$	$(-,2)$	$(-,2)$	$(-,2)$	$(-,2)$	$(2,-)$	$(-,2)$	$(2,2)$	$(2,2)$	$(2,2)$	(5)
$e_\pi(\mathsf{X}\mathsf{X}b,i)$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	?	?	?	
$d_\pi(\mathsf{X}a\mathsf{U}\mathsf{X}\mathsf{X}b,i)$	$(6,-)$	$(5,-)$	$(4,-)$	$(3,-)$	$(2,-)$	$(3,4)$	$(2,3)$	$(2,2)$	$(2,2)$	(6)
$e_\pi(\mathsf{X}a\mathsf{U}\mathsf{X}\mathsf{X}b,i)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	
$d_\pi(\mathsf{G}(\mathsf{X}a\mathsf{U}\mathsf{X}\mathsf{X}b),i)$	$(\infty,9)$	$(\infty,8)$	$(\infty,7)$	$(\infty,6)$	$(\infty,5)$	$(\infty,4)$	$(\infty,3)$	$(\infty,2)$	$(\infty,2)$	(7)
$e_\pi(\mathsf{G}(\mathsf{X}a\mathsf{U}\mathsf{X}\mathsf{X}b),i)$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	

Table 5.11: Traces of two systems that claim to implement $\mathsf{F}\mathsf{G}a \vee \mathsf{F}\mathsf{G}\neg a$.

	1	2	3	4	5	6	7	8	9	10	11	12	13
π_1: a	$\top$	$\top$	–	–	$\top$	$\top$	–	–	$\top$	$\top$	–	–	$\top$
π_2: a	$\top$	$\top$	–	–	$\top$	$\top$	–	–	$\top$	$\top$	$\top$	$\top$	$\top$

trace π_2 the truth value of a seems to remain stable from $i = 9$ onwards. Applying our approach, the first sequence (π_1) evaluates to presumably false because the suffix with one time $a = \top$ is shorter than a previous observed sequence of as being stable (e.g. at position $i = 1$ the truth value of a was stable for two time steps). In the second sequence, the suffix with five times $a = \top$ is longer than any previous sequence of as being stable and, thus, our approach evaluates this trace to presumably true.

These two examples also illustrate the importance of having a trace not truncated too early. Imagine cutting the trace at $i = 5$ or $i = 9$, then both traces evaluate to presumably false with respect to previously observed behavior, because we miss the observation of the long stable suffix.

When one subformula dominates: We now discuss a shortcoming of our approach. Consider the following specification

$$\phi = \mathsf{G}(\mathsf{F}a \vee \mathsf{F}b).$$

This specification requires that for any index i either signal a evaluates to true now or at a future position or, otherwise, signal b evaluates to true now

Table 5.12: Evaluation of $\mathsf{G}(\mathsf{F}a \vee \mathsf{F}b)$.

i	1	2	3	4	5	6	EOT
a	$\top$	$\top$	$\top$	$\top$	–	–	
b	$\top$	–	$\top$	–	$\top$	–	
$d_\pi(a,i)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,0)$
$e_\pi(a,i)$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	?
$d_\pi(\mathsf{F}a,i)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(2,\infty)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(\mathsf{F}a,i)$	$\top$	$\top$	$\top$	$\top$	$\bot_P$	$\bot_P$	$\top_P$
$d_\pi(b,i)$	$(0,-)$	$(-,0)$	$(0,-)$	$(-,0)$	$(0,-)$	$(-,0)$	$(0,0)$
$e_\pi(b,i)$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	?
$d_\pi(\mathsf{F}b,i)$	$(0,-)$	$(1,-)$	$(0,-)$	$(1,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(\mathsf{F}b,i)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top_P$	$\top_P$
$d_\pi(\mathsf{F}a \vee \mathsf{F}b,i)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(\mathsf{F}a \vee \mathsf{F}b,i)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot_P$	$\top_P$
$d_\pi(\mathsf{G}(\mathsf{F}a \vee \mathsf{F}b),i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)
$e_\pi(\mathsf{G}(\mathsf{F}a \vee \mathsf{F}b),i)$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\top_P$

or at a future position. In Table 5.12 we see that our approach concludes the trace under evaluation to presumably false. This is not what we would expect, as for positions smaller than or equal to 4, the formula $\mathsf{F}a$ is always satisfied immediately in the same time step and for all observed positions $i \leq 5$ the formula $\mathsf{F}b$ is satisfied within in at most one additional time step. In position $i = 6$ our approach predicts the formula $\mathsf{F}a \vee \mathsf{F}b$ to be presumably false, because the shorter witness for satisfaction dominates and, as both of the subformulas are eventually properties, none of them can be violated in finite time. Thus, the globally property is predicted to be violated which results in the evaluation of presumably false.

Intuitively, ϕ requires in every time step to eventually raise one of the two signals, i.e., one interpretation is that only the faster satisfaction counts. The specification $\phi' = \mathsf{GF}(a \vee b)$ is semantically equivalent to ϕ and expresses this interpretation formally and (also) evaluates to presumably false.

On the other side, if we rewrite ϕ to

$$\phi'' = \mathsf{GF}a \vee \mathsf{GF}b,$$

which is again semantically equivalent to ϕ, then the conclusion is presumably true (see Table 5.13), which is what we would expect. Thus, there is a difference in the interpretation of ϕ (and ϕ') and ϕ''. The specification ϕ'' can be interpreted such that the system only has to satisfy one of the two formulas $\mathsf{GF}a$ and $\mathsf{GF}b$, as those to formulas are connected with a logical or. Thus, the violation of one of the globally properties still allows the

Table 5.13: Evaluation of $\mathsf{GF}a \vee \mathsf{GF}b$.

i	1	2	3	4	5	6	EOT
a	$\top$	$\top$	$\top$	$\top$	–	–	
b	$\top$	–	$\top$	–	$\top$	–	
$d_\pi(a,i)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(-,0)$	$(-,0)$	$(0,0)$
$e_\pi(a,i)$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	?
$d_\pi(\mathsf{F}a,i)$	$(0,-)$	$(0,-)$	$(0,-)$	$(0,-)$	$(2,\infty)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(\mathsf{F}a,i)$	$\top$	$\top$	$\top$	$\top$	$\bot_P$	$\bot_P$	$\top_P$
$d_\pi(\mathsf{GF}a,i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)
$e_\pi(\mathsf{GF}a,i)$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\bot_P$	$\top_P$
$d_\pi(b,i)$	$(0,-)$	$(-,0)$	$(0,-)$	$(-,0)$	$(0,-)$	$(-,0)$	$(0,0)$
$e_\pi(b,i)$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	?
$d_\pi(\mathsf{F}b,i)$	$(0,-)$	$(1,-)$	$(0,-)$	$(1,-)$	$(0,-)$	$(1,\infty)$	$(0,\infty)$
$e_\pi(\mathsf{F}b,i)$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top_P$	$\top_P$
$d_\pi(\mathsf{GF}b,i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)
$e_\pi(\mathsf{GF}b,i)$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$
$d_\pi(\mathsf{GF}a \vee \mathsf{GF}b,i)$	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)	(∞,∞)
$e_\pi(\mathsf{GF}a \vee \mathsf{GF}b,i)$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$	$\top_P$

Table 5.14: Trace where evaluations differ for semantically equivalent specifications.

	1	2	3	4	5	6
a	$\top$	–	$\top$	–	–	–
b	–	–	–	$\top$	$\top$	$\top$

specification to be presumably satisfied (by the other globally).

Another example for two specifications that are semantically equivalent, but can be interpreted in different ways is:

$$\psi = \mathsf{G}(\mathsf{F}a \vee \mathsf{G}b)$$
$$\psi' = \mathsf{GF}a \vee (\mathsf{F}a\mathsf{UG}b)$$

While in specification ψ the formula $\mathsf{F}a$ dominates, because the formula $\mathsf{G}b$ cannot be satisfied in finite time, the rewriting to ψ' eliminates this dominating factor. Thus, for the trace presented in Table 5.14, evaluating ψ results in presumably false and evaluating ψ' results in presumably true.

System implements the specification in different modes: In the above examples we've shown a weakness of our approach that arises from a dominating subformula. The specifications with dominating subformulas for which our predictions fail have in common that they implicitly allow

systems to operate in two modes and (eventually) switch from one mode to the other.

Our approach may also fail for a system that operates in different modes when the mode is not part of the specification, e.g., a system that has a high- and a low-performance mode. Consider a system that implements the low-performance mode in such a way that the system takes longer to react (without violating the specification). When the trace contains system behavior of both modes, i.e., the high-performance and the low-performance mode, then our prediction is built on the behavior of the low-performance mode (assuming that witnesses are longer here), as we look at the longest observed witness for satisfaction. Thus, at some point predictions in the high-performance mode may be incorrect.

Shortcoming of our Approach: Consider the specification $\mathsf{GF}p$ and a system that raises p in the time steps $1, 2, 4, \ldots, 2^i$ with $i = 3 \ldots \infty$. As the distance for the next satisfaction of $\mathsf{F}p$ always doubles, we will give a wrong evaluation in half of the case. The reason for the wrong evaluation is that we have not yet observed witnesses with similar lengths for the second half of the last (doubled) distance to the (not yet observed) satisfaction of the eventually part.

Chapter 6

Conclusion and Outlook

> It is about simple awareness – awareness of what is so real and essential, so hidden in plain sight all around us, that we have to keep reminding ourselves, over and over: "This is water, this is water."
>
> David Foster Wallace

In this thesis we have presented approaches for automatic test case generation that help to bridge the gap between formal verification and testing. In this chapter we will summarize the presented work and draw conclusions in Section 6.1, before we give an outlook to possible future work in Section 6.2.

6.1 Summary and Conclusion

In companies that aim for certification of their products often two branches evolve in parallel. One branch aims for formal models and the other one is the usual development branch. Those two branches often have their own domain languages and, thus, to make use of knowledge from one domain in the other is difficult.

In this thesis we developed three tools for automatic test case generation that help to bridge the gap between those two branches. The presented approaches make use of existing formal models to automatically derive tests from them and also provide an approach that gives an indication on whether resulting finite traces of the SUT satisfy a given LTL property or not. A

test engineer, who's not familiar with formal methods, may use the presented tools to check the system under test for conformance to the specification and the (formalized) requirements. In the next three subsections we summarize our work.

6.1.1 Boolean Formulas

Transition guards may be (complex) formulas and exhaustive testing not possible. We implemented a tool that computes a test suite that achieves MCDC coverage on a given boolean formula. Our approach makes use of an SMT-solver that can also handle restrictions due to complex dependencies within the formula. This automatic test case generation method complements the certification in the development process as it can take the existing formal model and link the formal description of the product to the actual implementation.

We evaluated our approach on the applet firewall of an an implementation of the Java Card operating system. Our tool derived a small test suite and was able to improve the code coverage (condition + basic block coverage) of the existing test suite so that now all reachable locations and cases are covered. The additional tests produced by our tool also revealed that an update of the specification was not implemented. The MCDC criterion proved to be effective in our setting because it tests the different parts of the decisions in isolation without producing too many test cases.

For our case study of the Secure Block Device, simple node coverage already achieved a high line and branch coverage on the source code. While the test case generation time increased significantly for more complex coverage criteria, no gain in source code coverage was observed. As the model does not have any complex guards on the transitions, applying MCDC on the guards did not add any additional value. Executing the test suite that achieves simple node coverage, we found a real life bug in the SBD cache already. This illustrates that simple coverage criteria like node coverage may already yield test suites of sufficient quality to discover bugs that are hard to find manually.

6.1.2 Implementation Independent Tests

Using implementation specific knowledge for modeling implicitly influences tests derived from those models. We developed a tool that computes test strategies that aim for revealing a user defined class of faults in every almost correct implementation that claims to satisfy a given temporal logic

specification. The computed strategies do not rely on any implementation details. Our method is sound but incomplete, i.e., it may succeed finding a strategy but it may also fail finding one although a strategy exists. For many interesting cases, however, we have shown that it is both sound and complete.

The resulting strategies are generalized, such that the tester can assign values for inputs which do not influence the next state according to the specification. The user may also decide to compute another strategy that is different from the previous one. This opens many possible test cases for a specified class of faults.

We evaluated our tool on the AMBA specification, an industrial sized bus arbiter specification, as well as on the specification of an FDIR component. We executed the resulting strategies on real implementations and evaluated the code coverage as well as the ability to discover faults.

6.1.3 Runtime Verification Approach

Executing the strategy for a finite time on an reactive system results in a finite trace. While the system may not have obviously violated a property as the liveness part of the property can not get violated in finite time, the behavior can still be suspicious. We developed a semantics for evaluating LTL properties on finite traces. We assign for such an inconclusive suffix of a trace either presumably true, if the behavior of the system up to know gives rise that the property will be satisfied, and we assign presumably false for such an inconclusive suffix, if this behavior is worse than the behavior we have observed in the past.

To be able to asses an inconclusive suffix, we've presented a counting semantics that counts the number of steps to the next existing satisfaction, respectively violation, or otherwise indicates the earliest possible future satisfaction and violation. Based on this semantics we then evaluate inconclusive suffixes and assign truth values that indicate whether the trace is presumably true or presumably false.

We evaluated our semantics on properties and traces where existing semantics failed to provide a answer a human would expect.

6.2 Future Work

While this thesis presents approaches that support the tester by automatically deriving tests from existing formal models and also presents an ap-

proach for runtime verification of LTL properties, it also raises new questions that open up further research possibilities.

The test strategy generation from a temporal specification opens up different future research possibilities. The generalization of the strategy allows to follow different ideas. While we have implemented a proof of concept idea for generalization of the strategies, one can research ways to maximize the generalization or include additional intentions of the tester.

Implementations of a generalized test strategy opens up another field of research. While one may simply pick a random value for every input that is not fixed to a concrete value, one may also investigate more sophisticated approaches.

A general improvement can be to narrow down the choice for the input values. Remember the fundamental testing concepts presented in Chapter 2.3. The presented approaches do generate tests that fall into different equivalence classes with respect to the test purpose. The exact value is, however, picked automatically by the tool, as for example the SMT-solver, or left open to be chosen by the test engineer within specified bounds. Future work may focus on identifying bounds along the equivalence classes and automatically choosing values that are closer to these bounds than others. Such an improvement can then be added to the presented tools.

This thesis focuses on computing tests from existing models to check conformance of the system under test to the specification and the requirements, future work may use tests not only for conformance checking but also for learning a model of a blackbox system. This model can then be formally verified or manually investigated. The tool for adaptive test case generation may be of help to investigate the behavior of the blackbox.

The counting semantics presented in Chapter 5 opens the possibility to evaluate finite traces with respect to properties specified on infinite paths. While our definition of the semantics is based on a single trace, it is easily extended to take an entire set of traces into account instead and, thus, maybe provide a more precise prediction. Our approach uses a very simple form of learning to predict the future. Hence, investigating learning methods may be very promising for improving the prediction function.

While test case generation offers so many challenges and various ways to solve them, there is no way to ever get familiar with everything. It is, however, important to know and understand different approaches. To know what techniques are best used in which situations and to be able to ask the right questions at the right time.

Bibliography

[1] A. T. Acree et al. *Mutation Analysis*. Tech. rep. GIT-ICS-79/08. Atlanta, Georgia: Georgia Institute of Technology, 1979.

[2] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. "Validation, Verification, and Testing of Computer Software". In: *ACM Comput. Surv.* 14.2 (1982), pp. 159–192. DOI: 10.1145/356876.356879. URL: http://doi.acm.org/10.1145/356876.356879.

[3] B. K. Aichernig et al. "Killing strategies for model-based mutation testing". In: *Software Testing, Verification and Reliability* 25.8 (2015), pp. 716–748. DOI: 10.1002/stvr.1522. URL: http://dx.doi.org/10.1002/stvr.1522.

[4] G. Aleksandrowicz et al. "Designing reliable cyber-physical systems overview associated to the special session at FDL'16". In: *2016 Forum on Specification and Design Languages, FDL 2016, Bremen, Germany, September 14-16, 2016*. 2016, pp. 1–8. DOI: 10.1109/FDL.2016.7880382. URL: https://doi.org/10.1109/FDL.2016.7880382.

[5] B. Alpern and F. B. Schneider. "Defining Liveness". In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0. URL: https://doi.org/10.1016/0020-0190(85)90056-0.

[6] R. Alur, C. Courcoubetis, and M. Yannakakis. "Distinguishing tests for nondeterministic and probabilistic machines". In: *STOC'95*. 1995. DOI: 10.1145/225058.225161. URL: http://doi.acm.org/10.1145/225058.225161.

[7] P. Ammann, W. Ding, and D. Xu. "Using a Model Checker to Test Safety Properties". In: *ICECCS'01*. 2001, pp. 212–221. DOI: 10.1109/ICECCS.2001.930180. URL: http://dx.doi.org/10.1109/ICECCS.2001.930180.

[8] P. Ammann, A. J. Offutt, and H. Huang. "Coverage Criteria for Logical Expressions". In: *International Symposium on Software Reliability Engineering (ISSRE'03)*. IEEE, 2003, pp. 99–107.

[9] P. Ammann, J. Offutt, and W. Xu. "Coverage Criteria for State Based Specifications". In: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. 2008, pp. 118–156. DOI: 10.1007/978-3-540-78917-8_4. URL: http://dx.doi.org/10.1007/978-3-540-78917-8_4.

[10] R. Armoni et al. "Enhanced Vacuity Detection in Linear Temporal Logic". In: *CAV'03*. 2003, pp. 368–380. DOI: 10.1007/978-3-540-45069-6_35. URL: http://dx.doi.org/10.1007/978-3-540-45069-6_35.

[11] E. T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Trans. Software Eng.* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785. URL: http://dx.doi.org/10.1109/TSE.2014.2372785.

[12] C. Barrett, A. Stump, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2010.

[13] E. Bartocci et al. "A Counting Semantics beyond LTL Monitorability". In: *under review* (). under submission.

[14] E. Bartocci et al. "A Counting Semantics for Runtime LTL". In: *Journal* (Under Submission).

[15] A. Bauer, M. Leucker, and C. Schallhart. "Monitoring of Real-Time Properties". In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*. 2006, pp. 260–272. DOI: 10.1007/11944836_25. URL: http://dx.doi.org/10.1007/11944836_25.

[16] A. Bauer, M. Leucker, and C. Schallhart. "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?" In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. 2007, pp. 126–138. DOI: 10.1007/978-3-540-77395-5_11. URL: http://dx.doi.org/10.1007/978-3-540-77395-5_11.

[17] I. Beer et al. "Efficient Detection of Vacuity in ACTL Formulaas". In: *CAV'97*. 1997, pp. 279–290. DOI: 10.1007/3-540-63166-6_28. URL: http://dx.doi.org/10.1007/3-540-63166-6_28.

[18] I. Beer et al. “Efficient Detection of Vacuity in ACTL Formulaas”. In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*. 1997, pp. 279–290. DOI: 10.1007/3-540-63166-6_28. URL: http://dx.doi.org/10.1007/3-540-63166-6_28.

[19] G. Bernot, M. C. Gaudel, and B. Marre. “Software testing based on formal specifications: a theory and a tool”. In: *Software Engineering Journal* 6.6 (1991), pp. 387–405.

[20] G. Bernot, M. C. Gaudel, and B. Marre. “Software Testing Based on Formal Specifications: A Theory and a Tool”. In: *Softw. Eng. J.* 6.6 (Nov. 1991), pp. 387–405. ISSN: 0268-6961. DOI: 10.1049/sej.1991.0040. URL: http://dx.doi.org/10.1049/sej.1991.0040.

[21] D. Beyer et al. “Generating Tests from Counterexamples”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 326–335. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.999437.

[22] D. Beyer et al. “The software model checker BLAST”. In: *STTT* 9.5-6 (2007), pp. 505–525.

[23] A. Blass et al. “Play to Test”. In: *FATES'05*. LNCS 3997. Springer, 2005, pp. 32–46. ISBN: 3-540-34454-3. DOI: 10.1007/11759744_3. URL: http://dx.doi.org/10.1007/11759744_3.

[24] R. Bloem et al. “Automating Test-Suite Augmentation”. In: *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*. 2014, pp. 67–72. DOI: 10.1109/QSIC.2014.40. URL: http://dx.doi.org/10.1109/QSIC.2014.40.

[25] R. Bloem et al. “Case Study: Automatic Test Case Generation for a Secure Cache Implementation”. In: *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*. 2015, pp. 58–75. DOI: 10.1007/978-3-319-21215-9_4. URL: http://dx.doi.org/10.1007/978-3-319-21215-9_4.

[26] R. Bloem et al. “Interactive presentation: Automatic hardware synthesis from specifications: a case study”. In: *Design, Automation & Test in Europe (DATE'07)*. 2007, pp. 1188–1193. DOI: 10.1145/1266366.1266622. URL: http://doi.acm.org/10.1145/1266366.1266622.

[27] R. Bloem et al. "Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall". In: *VALID'13*. 2013, pp. 1–6.

[28] R. Bloem et al. "Synthesizing Adaptive Test Strategies from Temporal Logic Specifications". In: *Journal* (Under Submission).

[29] R. Bloem et al. "Synthesizing adaptive test strategies from temporal logic specifications". In: *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. 2016, pp. 17–24. DOI: 10.1109/FMCAD.2016.7886656. URL: https://doi.org/10.1109/FMCAD.2016.7886656.

[30] Board, IS. "IEEE Standard Classification for Software Anomalies". In: *IEEE Std* 1044 (2009).

[31] R. Bodík and B. Jobstmann. "Algorithmic program synthesis: introduction". In: *STTT* 15.5-6 (2013), pp. 397–411. DOI: 10.1007/s10009-013-0287-9. URL: https://doi.org/10.1007/s10009-013-0287-9.

[32] A. Bohy et al. "Acacia+, a tool for LTL synthesis". In: *Computer Aided Verification*. Springer. 2012, pp. 652–657.

[33] S. Boroday, A. Petrenko, and R. Groz. "Can a Model Checker Generate Tests for Non-Deterministic Systems?" In: *Electronic Notes in Theoretical Computer Science* 190.2 (2007), pp. 3–19. DOI: 10.1016/j.entcs.2007.08.002. URL: http://dx.doi.org/10.1016/j.entcs.2007.08.002.

[34] J. J. Chilenski. *An investigation of three forms of the modified condition decision coverage (MCDC) criterion*. Tech. rep. DTIC Document, 2001.

[35] J. J. Chilenski and S. P. Miller. "Applicability of modified condition/decision coverage to software testing". In: *Software Engineering Journal* 9.5 (1994), pp. 193–200. ISSN: 0268-6961.

[36] A. Church. "Logic, arithmetic and automata". In: *Proceedings of the international congress of mathematicians*. 1962, pp. 23–35.

[37] A. Cimatti et al. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. 2002, pp. 359–364. DOI: 10.1007/3-540-45657-0_29. URL: http://dx.doi.org/10.1007/3-540-45657-0_29.

[38] E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Workshop on Logics of Programs*. 1981, pp. 52–71. DOI: 10.1007/BFb0025774. URL: http://dx.doi.org/10.1007/BFb0025774.

[39] E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. 1981, pp. 52–71. DOI: 10.1007/BFb0025774. URL: http://dx.doi.org/10.1007/BFb0025774.

[40] E. M. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15. URL: http://dx.doi.org/10.1007/978-3-540-24730-2_15.

[41] V. D'Silva, D. Kroening, and G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 27.7 (2008), pp. 1165–1178. DOI: 10.1109/TCAD.2008.923410. URL: https://doi.org/10.1109/TCAD.2008.923410.

[42] A. David et al. "A Game-Theoretic Approach to Real-Time System Testing". In: *DATE'08*. 2008. DOI: 10.1109/DATE.2008.4484728. URL: http://dx.doi.org/10.1109/DATE.2008.4484728.

[43] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *IEEE Computer* 11.4 (1978), pp. 34–41. DOI: 10.1109/C-M.1978.218136. URL: http://dx.doi.org/10.1109/C-M.1978.218136.

[44] A. C. Dias Neto et al. "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*. ACM. 2007, pp. 31–36.

[45] L. K. Dillon and Y. S. Ramakrishna. "Generating Oracles from Your Favorite Temporal Logic Specifications". In: *SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996*. 1996, pp. 106–117. DOI: 10.1145/239098.239116. URL: http://doi.acm.org/10.1145/239098.239116.

[46] C. Eisner et al. "Reasoning with Temporal Logic on Truncated Paths". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. 2003, pp. 27–39. DOI: 10.1007/978-3-540-45069-6_3. URL: http://dx.doi.org/10.1007/978-3-540-45069-6_3.

[47] S. Falke, F. Merz, and C. Sinz. "The bounded model checker LLBMC". In: *ASE'13*. IEEE, 2013, pp. 706–709.

[48] B. Finkbeiner and S. Schewe. "Bounded synthesis". In: *STTT* 15.5-6 (2013), pp. 519–539. DOI: 10.1007/s10009-012-0228-z. URL: http://dx.doi.org/10.1007/s10009-012-0228-z.

[49] G. Fraser and P. Ammann. "Reachability and Propagation for LTL Requirements Testing". In: *QSIC'08*. 2008, pp. 189–198. DOI: 10.1109/QSIC.2008.21. URL: http://dx.doi.org/10.1109/QSIC.2008.21.

[50] G. Fraser and F. Wotawa. "Test-Case Generation and Coverage Analysis for Nondeterministic Systems Using Model-Checkers". In: *ICSEA'07*. 2007. DOI: 10.1109/ICSEA.2007.71. URL: http://dx.doi.org/10.1109/ICSEA.2007.71.

[51] G. Fraser, F. Wotawa, and P. E. Ammann. "Testing with Model Checkers: A Survey". In: *Softw. Test. Verif. Reliab.* 19.3 (Sept. 2009), pp. 215–261. ISSN: 0960-0833. DOI: 10.1002/stvr.v19:3. URL: http://dx.doi.org/10.1002/stvr.v19:3.

[52] F. Fummi and R. Wille. *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2016*. Vol. 454. Springer, 2017.

[53] P. L. Gall and A. Arnould. "Formal Specifications and Test: Correctness and Oracle". In: *Recent Trends in Data Type Specification, 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop, Oslo, Norway, September 19-23, 1995, Selected Papers*. 1995, pp. 342–358. DOI: 10.1007/3-540-61629-2_52. URL: https://doi.org/10.1007/3-540-61629-2_52.

[54] M. Gaudel. "Testing Can Be Formal, Too". In: *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*. 1995, pp. 82–96. DOI: 10.1007/3-540-59293-8_188. URL: https://doi.org/10.1007/3-540-59293-8_188.

[55] D. R. Graham. "Requirements and Testing: Seven Missing-Link Myths". In: *IEEE Software* 19.5 (2002), pp. 15–17. DOI: 10.1109/MS.2002.1032845. URL: https://doi.org/10.1109/MS.2002.1032845.

[56] K. Havelund and T. Pressburger. "Model Checking JAVA Programs using JAVA PathFinder". In: *STTT* 2.4 (2000), pp. 366–381.

[57] D. M. Hein, J. Winter, and A. Fitzek. "Secure Block Device - Secure, Flexible, and Efficient Data Storage for ARM TrustZone Systems". In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. 2015, pp. 222–229. DOI: 10.1109/Trustcom.2015.378. URL: http://dx.doi.org/10.1109/Trustcom.2015.378.

[58] R. M. Hierons. "Applying adaptive test cases to nondeterministic implementations". In: *Information Processing Letters* 98.2 (2006), pp. 56–60. DOI: 10.1016/j.ipl.2005.12.001. URL: http://dx.doi.org/10.1016/j.ipl.2005.12.001.

[59] R. M. Hierons et al. "Using Formal Specifications to Support Testing". In: *ACM Comput. Surv.* 41.2 (Feb. 2009), 9:1–9:76. ISSN: 0360-0300. DOI: 10.1145/1459352.1459354. URL: http://doi.acm.org/10.1145/1459352.1459354.

[60] A. Holzer et al. "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement". In: *CAV'08*. Vol. 5123. LNCS. Springer, 2008, pp. 209–213.

[61] M. Huth and M. D. Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)* Cambridge University Press, 2004.

[62] "ISO/IEC/IEEE Standard for Software Testing". In: (2013).

[63] S. M. Inc. *Java Card ^{TM} 2.1 Runtime Environment (JCRE) Specification*. 1999.

[64] S. M. Inc. *Java Card ^{TM} 2.2 Runtime Environment (JCRE) Specification*. 2006.

[65] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.

[66] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.

[67] P. C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition*. 3rd ed. AUERBACH, 2008. ISBN: 0849374758.

[68] H. Kelly J. et al. *A Practical Tutorial on Modified Condition/Decision Coverage*. Tech. rep. 2001.

[69] C. Kern and M. R. Greenstreet. "Formal verification in hardware design: a survey". In: *ACM Trans. Design Autom. Electr. Syst.* 4.2 (1999), pp. 123–193. DOI: 10.1145/307988.307989. URL: http://doi.acm.org/10.1145/307988.307989.

[70] A. Khalimov, S. Jacobs, and R. Bloem. "PARTY: Parameterized Synthesis of Token Rings". In: *CAV'13*. 2013, pp. 928–933. DOI: 10.1007/978-3-642-39799-8_66. URL: http://dx.doi.org/10.1007/978-3-642-39799-8_66.

[71] R. Könighofer, G. Hofferek, and R. Bloem. "Debugging formal specifications using simple counterstrategies". In: *FMCAD'09*. 2009. DOI: 10.1109/FMCAD.2009.5351127. URL: http://dx.doi.org/10.1109/FMCAD.2009.5351127.

[72] O. Kupferman and M. Y. Vardi. "Synthesis with incomplete informatio". In: *ICTL'97*. 1997, pp. 91–106.

[73] O. Kupferman and M. Y. Vardi. "Vacuity detection in temporal model checking". In: *STTT* 4.2 (2003), pp. 224–233. DOI: 10.1007/s100090100062. URL: http://dx.doi.org/10.1007/s100090100062.

[74] O. Kupferman and M. Y. Vardi. "Model Checking of Safety Properties". In: *Formal Methods in System Design* 19.3 (2001), pp. 291–314. DOI: 10.1023/A:1011254632723. URL: https://doi.org/10.1023/A:1011254632723.

[75] G. Luo, G. von Bochmann, and A. Petrenko. "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method". In: *IEEE Trans. Software Eng.* 20.2 (1994), pp. 149–162. DOI: 10.1109/32.265636. URL: http://dx.doi.org/10.1109/32.265636.

[76] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995. ISBN: 978-0-387-94459-3.

[77] D. A. Martin. "Borel determinacy". In: *Annals of Mathematics* 102.2 (1975), pp. 363–371.

[78] A. Morgenstern, M. Gesell, and K. Schneider. "An Asymptotically Correct Finite Path Semantics for LTL". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*. 2012, pp. 304–319. DOI: 10.1007/978-3-642-28717-6_24. URL: http://dx.doi.org/10.1007/978-3-642-28717-6_24.

[79] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Vol. 4963. LNCS. Springer, 2008, pp. 337–340.

[80] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN: 0471469122.

[81] L. Nachmanson et al. "Optimal strategies for testing nondeterministic systems". In: *ISSTA'04*. 2004, pp. 55–64. DOI: 10.1145/1007512.1007520. URL: http://doi.acm.org/10.1145/1007512.1007520.

[82] A. J. Offutt. "Investigations of the Software Testing Coupling Effect". In: *ACM Trans. Softw. Eng. Methodol.* 1.1 (1992), pp. 5–20. DOI: 10.1145/125489.125473. URL: http://doi.acm.org/10.1145/125489.125473.

[83] A. J. Offutt et al. "Generating test data from state-based specifications". In: *Softw. Test., Verif. Reliab.* 13.1 (2003), pp. 25–53. DOI: 10.1002/stvr.264. URL: http://dx.doi.org/10.1002/stvr.264.

[84] Oracle. *Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.4*. 2011.

[85] T. J. Ostrand and M. J. Balcer. "The category-partition method for specifying and generating fuctional tests". In: *Communications of the ACM* 31.6 (1988), pp. 676–686.

[86] A. Petrenko, A. da Silva Simão, and N. Yevtushenko. "Generating Checking Sequences for Nondeterministic Finite State Machines". In: *ICST'12*. 2012, pp. 310–319. DOI: 10.1109/ICST.2012.111. URL: http://dx.doi.org/10.1109/ICST.2012.111.

[87] A. Petrenko and A. Simão. "Generalizing the DS-Methods for Testing Non-Deterministic FSMs". In: *Computer Journal* 58.7 (2015), pp. 1656–1672. DOI: 10.1093/comjnl/bxu113. URL: http://dx.doi.org/10.1093/comjnl/bxu113.

[88] A. Petrenko and N. Yevtushenko. "Adaptive Testing of Nondeterministic Systems with FSM". In: *HASE'14*. 2014, pp. 224–228. DOI: 10.1109/HASE.2014.39. URL: http://dx.doi.org/10.1109/HASE.2014.39.

[89] A. Petrenko and N. Yevtushenko. "Conformance Tests as Checking Experiments for Partial Nondeterministic FSM". In: *FATES'05*. 2005, pp. 118–133. DOI: 10.1007/11759744_9. URL: http://dx.doi.org/10.1007/11759744_9.

[90] A. Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: http://dx.doi.org/10.1109/SFCS.1977.32.

[91] A. Pnueli and R. Rosner. "On the Synthesis of a Reactive Module". In: *Principles of Programming Languages (POPL'89)*. 1989, pp. 179–190. DOI: 10.1145/75277.75293. URL: http://doi.acm.org/10.1145/75277.75293.

[92] Radio Technical Commission for Aeronautics (RTCA). *RTCA-DO-178B: Software Considerations in Airbone Systems and Equipment Certification*. Dec. 1992.

[93] C. R. Spitzer and C. Spitzer. *Digital Avionics Handbook*. CRC press, 2000.

[94] L. Tan, O. Sokolsky, and I. Lee. "Specification-based Testing with Linear Temporal Logic". In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, November 8-10, 2004, Las Vegas Hilton, Las Vegas, NV, USA*. 2004, pp. 493–498. DOI: 10.1109/IRI.2004.1431509. URL: http://dx.doi.org/10.1109/IRI.2004.1431509.

[95] W. Thomas. "Automata and reactive systems". In: *Course Lectures, RWTH Aachen, Germany* (2002).

[96] J. Tretmans. "Model Based Testing with Labelled Transition Systems". In: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. 2008, pp. 1–38. DOI: 10.1007/978-3-540-78917-8_1. URL: http://dx.doi.org/10.1007/978-3-540-78917-8_1.

[97] G. M. Weinberg. *Perfect Software: And Other Illusions About Testing*. New York, NY, USA: Dorset House Publishing Co., Inc., 2008. ISBN: 0932633692, 9780932633699.

[98] E. Weyuker, T. Goradia, and A. Singh. "Automatically generating test data from a Boolean specification". In: *IEEE Transactions on Software Engineering* 20.5 (1994), pp. 353–363.

[99] M. W. Whalen et al. "Coverage metrics for requirements-based testing". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. 2006, pp. 25–36. DOI: 10.1145/1146238.1146242. URL: http://doi.acm.org/10.1145/1146238.1146242.

[100] J. Woodcock et al. "Formal methods: Practice and experience". In: *ACM Comput. Surv.* 41.4 (2009), 19:1–19:36. DOI: 10.1145/1592434.1592436. URL: http://doi.acm.org/10.1145/1592434.1592436.

[101] M. Yannakakis. "Testing, Optimizaton, and Games". In: *Logic in Computer Science (LICS'04)*. 2004, pp. 78–88. DOI: 10.1109/LICS.2004.1319602. URL: http://dx.doi.org/10.1109/LICS.2004.1319602.